Universidad de Valladolid


UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

# Librería para controlar un Reloj de tiempo real mediante Arduino.

Autor:

Vera Gallego, Pedro José

Tutor:

Plaza Pérez, Francisco José
Tecnología Electrónica

Valladolid, julio de 2017.

# Resumen

En el presente proyecto se ha diseñado e implementado una librería para el control de relojes de tiempo real (RTC) mediante Arduino.

Gran cantidad de RTC pueden controlarse con esta librería, centrada en el DS3231, pero válida para otros relojes que se comuniquen mediante $I^2C$.

La librería cuenta con un archivo de cabecera donde se definen las variables y funciones de la librería, y un archivo cpp donde se implementa el código de dichas funciones en lenguaje C++. Además, incluye un fichero keyword.txt con las palabras reservadas por la librería, variables, constantes y funciones.

Como en cualquier buena librería se añade una carpeta de ejemplos programados en Arduino. Un ejemplo para ajustar y ver la fecha y hora, otro ejemplo de prueba de alarmas y, por último, se ha implementado un programa para tratar un problema más industrial, la detección del cambio de tarifas eléctricas según la fecha y hora.

**Palabras clave:** Arduino – Microcontrolador – RTC - $I^2C$ - Interrupciones

# Abstract

In the present Project has been designed and implemented a library for the control of real-time clocks (RTC) using Arduino.

A lot of RTC devices could be controlled by this library, centered on the DS3231, but valid with other clocks comunicated through $I^2C$.

The librery has a header file, where variables and functions of the library are defined, and a cpp file where the code of those functions is implemented in C++ language. In addition, it includes a file keyword.txt with reserved words by the library, variables, constants and functions.

As in any good library is added a folder of Arduino sample programs. An example to set and watch the date and time, another for alarm test and a program to detection of electric rates according to the date an time.

**Keywords:** Arduino – Microcontroller – RTC - $I^2C$ - Interruptions

# Índice

# 1- Introducción y objetivo

## 1.1- Introducción:

El presente proyecto aborda el tema de los relojes de tiempo real (RTC), los cuales son muy útiles en gran cantidad de sistemas que requieren exactitud temporal, como puede ser un proceso que presenta restricciones temporales a la hora de iniciar o finalizar una tarea.

Para abordar el problema, será necesario conocer el entorno Arduino, así como el funcionamiento de los RTC y su utilidad. De todo esto se habla brevemente en el capítulo 2, ya que el proyecto se centra en controlar un RTC mediante un microcontrolador Arduino.

Además del entorno Arduino, se explica el protocolo $I^2C$, con el que se realiza la comunicación serie entre microcontrolador y RTC.

Ya que suele ser complicado leer código ajeno, aunque se ha tratado de usar variables auto-explicativas, en esta memoria se aclara el diseño y desarrollo de las funciones para controlar el RTC.

## 1.2- Objetivo:

El objetivo de este proyecto es la elaboración de los archivos necesarios para el funcionamiento de una librería Arduino para el control, ajuste y lectura de un reloj de tiempo real.

La librería incluye las funciones necesarias para la comunicación entre el microcontrolador Arduino y el reloj de tiempo real DS3231, mediante el protocolo $I^2C$, pero también es válida para la mayoría de RTC de esta familia, ya que la estructura de los registros es la misma y también usan la comunicación serie $I^2C$. Aunque algunas de las funciones pueden no ser compatibles con otros modelos, como pueden ser las funciones de las alarmas o la de solicitar temperatura, en caso de que dichos modelos no tengan control de alarma ni sensor de temperatura.

Para que sea sencillo conocer y probar el funcionamiento de las distintas funciones, se realizan varios programas de ejemplo. Primero uno básico de ajuste y lectura de la hora y un segundo programa para establecer alarmas y comprobar la activación de dichas alarmas

mediante un LED. Por último, se realiza un programa de un ámbito más industrial, este consistirá en gestionar las alarmas necesarias para cambiar el estado de una variable "tarifa" que representa la tarifa eléctrica dependiente de la fecha y la hora del día.

## 1.3- Estructura de la memoria:

En este apartado se resume brevemente los distintos puntos en que se divide la memoria:

- Capítulo 1. Introducción. Se realiza una introducción del tema del proyecto y se aclaran los objetivos del proyecto.

- Capítulo 2. Estado de la técnica. En este apartado se explican varios conceptos a conocer para el entendimiento del contenido del proyecto, como información sobre Arduino, el protocolo $I^2C$ o los dispositivos RTC.

- Capítulo 3. Diseño de la librería. Se presenta el diseño de cada función, con su diagrama de flujo, y las relaciones entre las distintas funciones.

- Capítulo 4. Implementación de las funciones. Una vez definido el diseño de las funciones y su conexión, en este punto se detalla el contenido y funcionamiento de cada una de ellas.

- Capítulo 5. Diseño y desarrollo de la aplicación de ejemplo.
  Este capítulo presenta el problema de gestión de las tarifas eléctricas y plantea el diseño e implementación de una posible solución.

- Capítulo 6. Conclusiones y líneas futuras. Se destaca los aspectos más relevantes del proyecto y las posibles líneas futuras.

- Capítulo 7. Bibliografía. Referencias empleadas.

- Capítulo 8. Anexos.

# 2- Estado de la técnica:

## 2.1- Arduino:

Arduino es una compañía de hardware libre, es decir, sus especificaciones y diagramas esquemáticos son de acceso público (Anexo 1). Arduino se dedica al diseño y manufactura de placas de desarrollo de hardware y software que integran un microcontrolador y su entorno de desarrollo.

Arduino se centra en facilitar el uso de la electrónica y programación de sistemas embebidos en proyectos de muy diversos tipos. Los inicios de Arduino se remontan a 2005, cuando comenzó a crecer gracias a su bajo costo y facilidad de uso para realizar prototipos de proyectos o interactuar con software instalado en el ordenador.

### 2.1.1- Hardware:

El hardware de un Arduino consiste en una placa de circuito impreso con un microcontrolador, normalmente de tipo Atmel AVR, una serie de entradas y salidas digitales y analógicas.

Se han desarrollado gran cantidad de modelos de placa Arduino, siendo algunos de los más conocidos el Arduino Uno, Arduino Leonardo, Arduino Mega o Arduino Yún.

El presente proyecto, aunque es una librería, la cual es compatible con cualquier modelo de Arduino, se ha probado en Arduino Uno, por lo que en los programas de ejemplo los pines se corresponden a los de dicha placa. Siendo importante conocer que los pines de interrupciones para Arduino Uno son el pin 2 y pin 3, para las interrupciones 0 y 1 respectivamente. La configuración de pines se puede ver en el Anexo 2.

Características del Arduino Uno:

- Microcontrolador: ATmega328
  - Voltaje: 5V
  - Voltaje de entrada (recomendado): 7-12V
  - Voltaje de entrada (limites): 6-20V
  - Pines digitales I/O: 14 (de los cuales 6 son salida PWM)
  - Entradas Analógicas: 6

- Corriente DC por I/O Pin: 40 mA
- Corriente DC para 3.3V Pin: 50 mA
- Memoria Flash: 32 KB (ATmega328) de los cuales 0.5 KB son utilizados para el arranque
- SRAM: 2 KB (ATmega328)
- EEPROM: 1 KB (ATmega328)
- Velocidad del reloj: 16 MHz

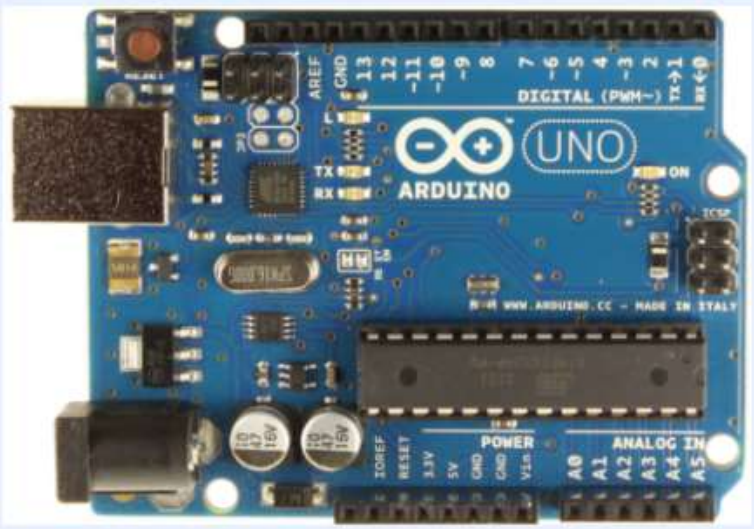



Figura 1. Placa Arduino UNO.

### 2.1.2- Lenguaje:

La programación de Arduino se realiza mediante un lenguaje propio, el cual está basado en el lenguaje de programación de alto nivel Processing, similar a C++, por lo que todas las funciones del lenguaje estándar C y algunas de C++ las soporta la plataforma Arduino.

### 2.1.3- Librerías:

La creación y utilización de librerías favorece mucho el diseño y organización de proyectos, ya que es una manera de separar el código que contiene funciones generales, las cuales se pueden usar en distintos proyectos, del código fuente principal del proyecto.

Existen librerías propias de Arduino, las cuales se usarán en este proyecto, como la librería Wire, que permite el envío y recepción de datos mediante protocolo I2C, de la que se habla en el siguiente punto, o la librería Serial, que permite la lectura y escritura por el puerto serie.

También están las librerías creadas por los usuarios, como la del presente proyecto para controlar un reloj de tiempo real. Para que una librería creada sea compatible con Arduino, simplemente debe tener los siguientes archivos:

- Fichero de cabecera, con extensión .H, en el que se declaran las funciones y variables utilizadas por la librería.
- Fichero de programa, con extensión .cpp, en donde se encuentra el código de las funciones, tanto públicas como privadas.
- Fichero *keywords*, es un fichero de texto donde se definen las palabras clave para llamar desde Arduino a las funciones o variables de la librería. Dichas palabras se reservan y al escribirlas en Arduino cambian de formato.
- Opcionalmente suele haber una carpeta con ficheros de extensión .ino en los que se utilicen funciones de la librería a modo de ejemplo.

## 2.2- Dispositivos RTC:

Un reloj en tiempo real, RTC (del inglés real time clock), es un dispositivo que es capaz de actualizar la hora por si solo, iniciándose en la hora a la que se programe, y siguiendo el sistema temporal en unidades humanas, es decir, años, meses, días, horas, minutos y segundos.

Estos dispositivos se incluyen en un circuito integrado y se emplean en la mayoría de aparatos electrónicos que requieren almacenar el tiempo exacto, como ordenadores personales, servidores o sistemas empotrados.

Una de las principales ventajas que tiene el utilizar estos dispositivos, es liberar al sistema principal de trabajo, esto unido a su precisión y al bajo consumo de energía, hace que los RTC sean muy útiles y se recurra a ellos cuando se necesite controlar tiempo. Además, los RTC poseen una fuente de alimentación propia que les permite seguir midiendo el tiempo mientras no recibe alimentación de la fuente principal. Antiguamente esta alimentación alternativa era una batería de litio, pero los sistemas más modernos usan un supercapacitor, ya que son recargables y pueden ser soldados a la placa.
Para realizar la medición de tiempo la mayoría de relojes utilizan un oscilador de cristal, aunque algunos usan la propia frecuencia de la fuente de alimentación. La frecuencia del oscilador es de 32.768 kHz, $2^{15}$ ciclos por segundo.

Uno de los mayores fabricantes de RTC es la compañía Dallas, la cual diseñó una familia de integrados denominada DS en la que se ha centrado este proyecto, por lo que la mayoría de los módulos de esta familia se puede controlar con la librería desarrollada. En concreto, para la realización y prueba de esta librería se ha empleado el reloj DS3231, descrito a continuación.

## DS3231

Este dispositivo es un RTC de bajo costo y extremadamente preciso, el cual cuenta con los anteriormente mencionados, oscilador de cristal y batería alternativa, todas sus características se pueden comprobar en el datasheet, incluido en el Anexo 3.

Este RTC ajusta automáticamente los meses que tienen menos de 31 días, incluyendo febrero, con su corrección en año bisiesto. Puede trabajar en los dos formatos horarios, tanto el de 24 horas, como el de 12 horas con un indicador AM/PM. Posee dos alarmas programables en día y hora. La dirección y datos se transfieren en serie mediante un bus bidireccional $I^2C$. Además, mediante una referencia de tensión compensada por temperatura y un circuito comparador, controlan el estado de la alimentación, para detectar fallos en la misma, para producir un reinicio y cambiar automáticamente a la fuente de alimentación de reserva.

La descripción de los pines del integrado es:

- Pin 1: salida del oscilador de 32kHz, requiere una resistencia externa de pull-up, pero en caso de no ser usado puede dejarse libre.
- Pin 2: VCC, pin de la alimentación primaria. Debe ser desacoplado con un condensador de 0.1 µF a 1 µF. Si no se usa, se conecta a tierra.
- Pin 3: INT/SQW, a nivel bajo activa las interrupciones. Requiere una resistencia de pull-up conectada a una alimentación de 5.5V o menor. Este pin está determinado por el estado del bit INTCN del registro de Control. Si es un 0 lógico, la salida del pin es una onda cuadrada cuya frecuencia depende de los bits RS2 y RS1 del registro de Control. Si es un 1 lógico, entonces cuando una alarma se active, la salida indicará una interrupción poniéndose a nivel bajo. Si no se usa puede quedar desconectado.
- Pin 4: RST a nivel bajo, este pin es de entrada salida e indica el estado de la alimentación VCC en relación con el voltaje VPF. Si VCC cae por debajo de VPF, el pin RST pasa a nivel bajo, si VCC excede el

valor de VPF, el pin RST pasará a nivel alto, debido a la resistencia pull-up interna. No deben conectarse resistencias externas de pull-up.

- Pines de 5 al 12: conectados a tierra.
- Pin 13: Tierra.
- Pin 14: VBAT, es la entrada de la alimentación de reserva. Si se está utilizando la alimentación principal, este pin debe desacoplarse con condensador de baja fuga de 0.1 µF a 1 µF. Si no se utiliza, se debe conectar a tierra.
- Pin 15: *SDA*, es el pin de entrada y salida de datos en serie. Requiere una resistencia externa de pull-up.
- Pin 16: *SCL*, es el pin de reloj para el interfaz serie I²C, el cual sincroniza los movimientos de los datos.

El diagrama de bloques de este dispositivo es el siguiente:

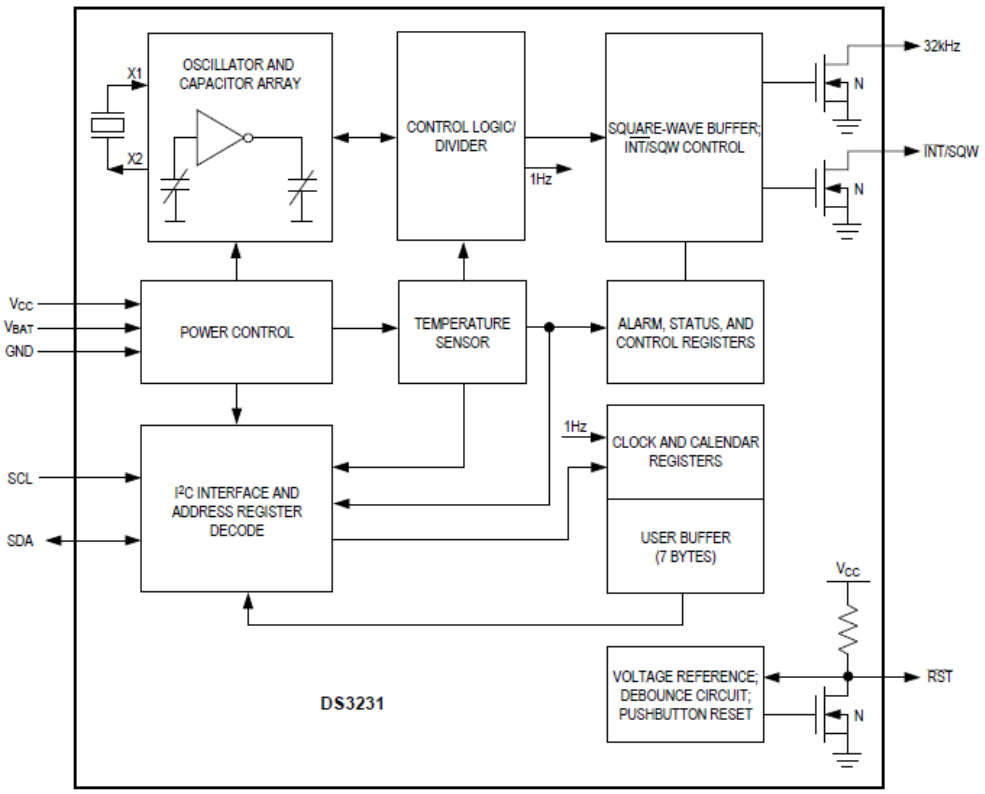



Figura 2. Diagrama de bloques del DS3231.

El resto de características se detallarán junto a la implementación del código de la librería, para comprender mejor el sentido de las distintas funciones que trabajan con los registros de hora y fecha, de alarmas, temperatura, control o estado.

## 2.3- Comunicación serie mediante I²C:

I²C es la abreviatura de "Inter-Integrated Circuit", una técnica que presentaron los ingenieros de "Philips", en 1982, como solución a la necesidad de simplificar y normalizar las líneas de intercambio de datos entre distintos circuitos integrados. Este sistema reduce el número de cables a dos, uno de datos, *SDA*, y otro de reloj, *SCL*, ambas conectadas a VDD mediante una resistencia pull-up. El bus trabaja con lógica positiva, donde el nivel alto debe ser al menos 0,7 veces VDD y el nivel bajo no debe ser mayor de 0,3 veces VDD.

El bus I²C está diseñado como un bus maestro-esclavo, donde el maestro inicia la transferencia de datos y el esclavo reacciona. Lo normal es que haya un maestro y múltiples esclavos, pero también existe la posibilidad de controlar el acceso al bus, haciendo posible tener varios maestros que pueden ir turnándose. Para este proyecto, el maestro será el microcontrolador de Arduino y el esclavo el módulo RTC, pudiéndose añadir un esclavo más, que sería un display *LCD*, para visualizar los valores del reloj.

El maestro genera la señal de reloj (*SCL*), la cual tiene permitido un pulso máximo dependiendo del modo en que trabaje, siendo posible utilizar señales de reloj más lentas cuando el maestro lo permita. En caso de que el esclavo necesite más tiempo del que le permite el maestro, la señal de reloj se puede mantener a nivel bajo para frenar al maestro, entre la transferencia de bytes individuales. Los datos de estos bytes individuales solo son válidos si su nivel lógico no varía durante la fase de reloj a nivel alto, a excepción del inicio, parada y el reset. Una señal de arranque es un flanco descendente en *SDA* mientras *SCL* está a nivel alto, la de parada es un flanco ascendente en *SDA* mientras *SCL* está a nivel alto y la señal de reset tiene el mismo comportamiento que la de inicio.

Los datos constan de un byte (8 bits, de dirección o con un valor, dependiendo del protocolo) y un bit de *ACK* (de confirmación), que se transfieren con el bit más significativo primero. Este bit lo pondrá a nivel alto un esclavo durante un nivel bajo de *SDA* en el noveno nivel alto de *SCL*. El esclavo debe poner a nivel bajo *SDA* antes de que cambie *SCL* a nivel alto, para que otros dispositivos del bus no lo interpreten como una señal de arranque.

La dirección de I²C estándar es el primer byte de datos que el maestro envía, siendo los 7 primeros bits la dirección y el octavo el bit de lectura o escritura, que indica si el esclavo debe recibir datos, si está a nivel bajo, o enviarlos, a nivel alto. Por lo que este protocolo permite 128 direcciones, aunque solo se podrán usar libremente 112, ya que hay 16 nodos reservados para fines especiales. La dirección de cada circuito integrado viene definida por el fabricante, aunque los últimos 3 bits pueden ser fijados para *tener* una segunda dirección, esto sirve en caso de que se tengan varias direcciones idénticas, por ejemplo, si se tienen varios dispositivos iguales, lo que permitiría tener hasta 8 integrados con la misma dirección. Más tarde se introdujo un direccionamiento de 10 bits, compatible con el estándar de 7 bits.

Como ya se ha comentado, la transferencia de datos es iniciada por el maestro con una señal de arranque y seguida de una señal con la dirección. El esclavo confirma la llegada de la dirección mediante el bit *ACK* y en función del bit R/W se escriben datos en el esclavo o se leen de él, del mismo modo, al escribir, el ACK será enviado por el esclavo y al leer por el maestro, reconociendo este el último byte leído como un *NACK*, que indica el final de la transmisión. Para finalizar la transmisión se envía una señal de parada, pero es posible enviar una señal de reset en caso de que se quiera volver a transmitir, sin necesidad de enviar la señal de parada.

En el entorno Arduino existe una librería para controlar dispositivos conectados mediante este bus. Esta es la librería Wire, de la cual se hablará en el desarrollo del programa.

Para concluir esta breve explicación sobre el bus I²C, diremos que es un protocolo bastante simple, pero muy susceptible a las interferencias, por lo que se emplea en comunicaciones que no tengan ruido ni distancias largas, como puede ser la conexión de integrados en una misma placa, que no tenga diafonías ni problemas de acoplamiento electromagnético.

# 3- Diseño de la librería

En este capítulo se presenta la estructura elegida para la librería. Para ello se mostrarán los diagramas de flujo de las diferentes funciones, así como los diagramas de bloques que muestran la relación entre las mismas.

## 3.1- Estructura de la librería:

La librería se ha diseñado con una serie de funciones públicas, que son las que utilizará el usuario para controlar el RTC en el código fuente del programa Arduino que desee realizar, y un apartado de funciones privadas, las cuales usan internamente las funciones anteriores, o incluso estas mismas funciones privadas. Siendo algunas de las más importantes y por tanto las más llamadas dentro de otras funciones, la función pública *"leerHora"*, para leer los registros del reloj y almacenarlos en una variable con la estructura de datos como la del RTC, y la privada *"escribeReloj"*, para escribir nuevos datos o sustituir los anteriores en los registros del reloj.

Para comprender la utilidad de las distintas funciones y su código, es necesario conocer la Tabla 1, la cual muestra los registros del RTC (en este caso del DS3231). Tabla que se debe tener en cuenta también en la implementación.

| ADDRESS | BIT 7 MSB | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 LSB | FUNCTION | RANGE |
|---|---|---|---|---|---|---|---|---|---|---|
| 00h | 0 | 10 Seconds | | | Seconds | | | | Seconds | 00–59 |
| 01h | 0 | 10 Minutes | | | Minutes | | | | Minutes | 00–59 |
| 02h | 0 | 12/$\overline{24}$ | $\overline{AM}$/PM<br>20 Hour | 10 Hour | Hour | | | | Hours | 1–12 + $\overline{AM}$/PM<br>00–23 |
| 03h | 0 | 0 | 0 | 0 | 0 | Day | | | Day | 1–7 |
| 04h | 0 | 0 | 10 Date | | Date | | | | Date | 01–31 |
| 05h | Century | 0 | 0 | 10 Month | Month | | | | Month/ Century | 01–12 + Century |
| 06h | 10 Year | | | | Year | | | | Year | 00–99 |
| 07h | A1M1 | 10 Seconds | | | Seconds | | | | Alarm 1 Seconds | 00–59 |
| 08h | A1M2 | 10 Minutes | | | Minutes | | | | Alarm 1 Minutes | 00–59 |
| 09h | A1M3 | 12/$\overline{24}$ | $\overline{AM}$/PM<br>20 Hour | 10 Hour | Hour | | | | Alarm 1 Hours | 1–12 + $\overline{AM}$/PM<br>00–23 |
| 0Ah | A1M4 | DY/DT | 10 Date | | Day | | | | Alarm 1 Day | 1–7 |
| | | | | | Date | | | | Alarm 1 Date | 1–31 |
| 0Bh | A2M2 | 10 Minutes | | | Minutes | | | | Alarm 2 Minutes | 00–59 |
| 0Ch | A2M3 | 12/$\overline{24}$ | $\overline{AM}$/PM<br>20 Hour | 10 Hour | Hour | | | | Alarm 2 Hours | 1–12 + $\overline{AM}$/PM<br>00–23 |
| 0Dh | A2M4 | DY/$\overline{DT}$ | 10 Date | | Day | | | | Alarm 2 Day | 1–7 |
| | | | | | Date | | | | Alarm 2 Date | 1–31 |
| 0Eh | $\overline{EOSC}$ | BBSQW | CONV | RS2 | RS1 | INTCN | A2IE | A1IE | Control | — |
| 0Fh | OSF | 0 | 0 | 0 | EN32kHz | BSY | A2F | A1F | Control/Status | — |
| 10h | SIGN | DATA | DATA | DATA | DATA | DATA | DATA | DATA | Aging Offset | — |
| 11h | SIGN | DATA | DATA | DATA | DATA | DATA | DATA | DATA | MSB of Temp | — |
| 12h | DATA | DATA | 0 | 0 | 0 | 0 | 0 | 0 | LSB of Temp | — |

Tabla 1. Registros del dispositivo DS3231.

En esta tabla se muestra la estructura de los registros del RTC, cuya dirección es la de la columna de más a la izquierda, empezando por 00h, hasta llegar a la 12h, es decir, del 0 al 18 en el sistema decimal. Se muestra también que el bit más significativo de cada registro es el bit 7, siendo por tanto el 0 el menos significativo. Se pueden distinguir 3 bloques de registros:

- Los siete primeros registros, del 00h al 06h, corresponden a los datos de hora y fecha, donde el bit más significativo es 0, exceptuando el registro del mes, que contiene el siglo (no se usará, ya que es innecesario), y el registro del año, ya que puede llegar hasta 99, por lo que precisa el bit 7.
- Los siguientes siete, del 07h al 0Dh, contienen las alarmas, el bit más significativo es el bit de máscara de alarma, dependiendo de cuales de ellos estén a nivel alto o a nivel bajo, el reloj tendrá un tipo de alarma activa u otra. La tabla que muestra las distintas configuraciones se mostrará más adelante. El resto de bits, como en el caso de los anteriores registros, contendrá el dato de hora o fecha, pero en este caso no el que irá incrementando el reloj, si no el de la alarma establecida. Distinguir también que los cuatro primeros registros corresponden a la alarma 1 y los tres siguientes a la alarma 2.
- Los últimos cinco registros también podemos catalogarlos en 3 distintos bloques, donde los registros 0Eh y 0Fh son de control y estado respectivamente, se emplearán para el control de las alarmas. Si el bit INTCN y alguno o los dos bits A2IE y A1IE están a nivel alto, las alarmas estarán activas, permitiendo que los bits A2F y A1F se pongan a nivel alto en caso de haber una coincidencia entre la hora de los registros del primer bloque y la alarma establecida en el segundo. El registro 10h, contiene los datos de offset, que no se utilizarán en esta librería. Los dos registros finales, 11h y 12h, son los que contendrán los datos leídos por el sensor de temperatura.

Conocidos estos registros y sabiendo que los datos se almacenan en hexadecimal, podemos definir las funciones necesarias para el funcionamiento de la librería, que se agrupan en dos clases, una primera para la estructura de los datos almacenados, la clase Reloj, y otra denominada RTC que contiene las siguientes funciones:

- Función para realizar la lectura de los registros de tiempo, *"leerHora"*.
- Una función para decodificar, pasando de hexadecimal a decimal, y otra de codificación, haciendo la operación inversa, decodificar y codificar.
- Una función que escriba los nuevos valores en los registros de tiempo, previa codificación, *"escribeReloj"*.

- Funciones para cambiar los valores de los registros de tiempo, *"escribeHora"*, *"escribeFecha"*, *"escribeDia"*.
- Es necesaria una función para modificar el registro de control y de este modo activar y cambiar el tipo de alarma, *"alarmasON"*.
- Para definir el instante en que se activará la alarma se debe añadir a los registros de forma similar que en los de tiempo, *"escribeAlarma"*.
- En caso de que se haya producido una alarma, se debe reestablecer a nivel bajo los bits AF2 y AF1 del registro de estado, *"limpiaAlarma"*.
- Será necesaria una función para elegir si se quiere almacenar y leer la hora en formato de doce o veinticuatro horas, *"Modo12h"*.
- Para comprobar durante la ejecución del programa en que modo se está trabajando, se utilizará la función *"am_pm"*, y la función bit12_24, que asegura que el bit de este modo esté correcto, además de comprobar, en caso de ser modo doce horas, el valor del bit AM/PM, almacenándolo en una variable.
- Otras de las funciones necesarias para el cambio de modo son las que transforman la hora en formato doce horas a veinticuatro y viceversa, llamadas "*de12a24*" y "*de24a12*".
- A la hora de llamar a *"escribeReloj"*, se debe comprobar si los datos a escribir están dentro de los límites, por ejemplo, los segundos o minutos deben estar entre 0 y 60. Para estas comprobaciones se usan las funciones booleanas *"datoEntre"*, como *"horaEntre"* o *"diaEntre"*.
- Dos funciones, *"leeRegistro"* y *"escribeRegistro"*, como su nombre indica, sirven para la lectura y escritura de un registro en concreto.
- Para pasar a una cadena de caracteres y poder mostrar la fecha y hora al usuario, se utilizarán las funciones *"verHora"*, *"verFecha"*, *"verMes"* y *"verDia"*.
- Las funciones que convierten en cadena de caracteres los datos usarán una función que comprueba si el dato es de una cifra o de dos, *"digitosDato"*.
- La función *"verFecha"* en concreto, puede solicitarse en tres formatos distintos en el orden de los días, meses y años. Para reordenar la cadena se utilizarán las funciones cambio e invierte.
- Cuando se introduce la fecha, el día de la semana también se almacena, ya que se llama a una función qué conocido el día, mes y año, calcula a que día de la semana corresponde, *"calculaDiaSem"*.
- Con la idea de poder cambiar el contenido de los registros con señales digitales, se introducen las funciones *"variaDatos"*, como *"variaHora"* o *"variaDia"*, las cuales incrementan o decrementan los valores en una unidad.
- Por último, es necesaria una función que compruebe si es necesario reajustar la hora, debido a un cambio horario, *"compruebaCH"*.

## 3.2- Relación entre funciones:

A continuación, se muestran los diagramas de bloques que representan las conexiones entre las distintas funciones (figuras de la 3 a la 23). Solo se presentan los diagramas de funciones que realizan alguna llamada a otras funciones, es decir, que precisan de otra función para llevase a cabo.

En la parte superior de cada diagrama, en verde, se sitúa la función principal, debajo, en azul, le suceden las funciones a las que se llama dentro de ella. En caso de que la función principal devuelva un valor a otra función, se indica con una flecha, y de nuevo en verde la función a la que se envía dicho dato.

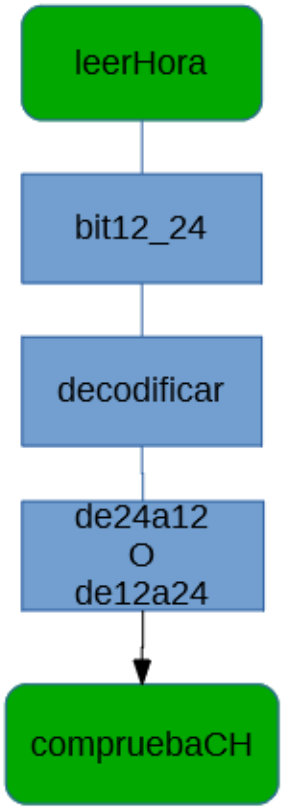



Figura 3.

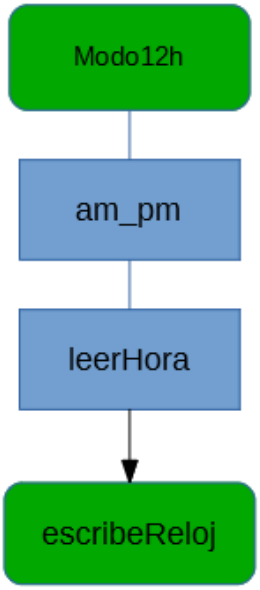



Figura 4.

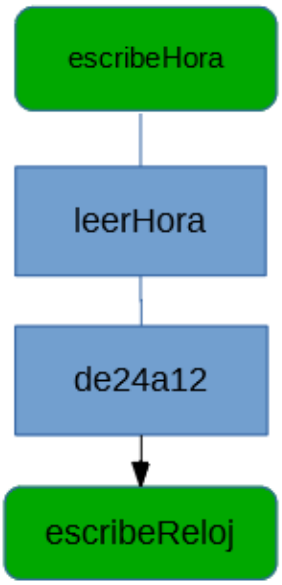



Figura 5.

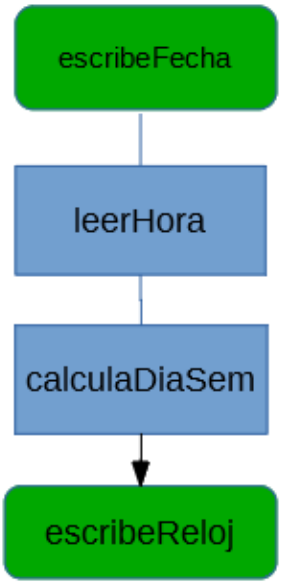



Figura 6.

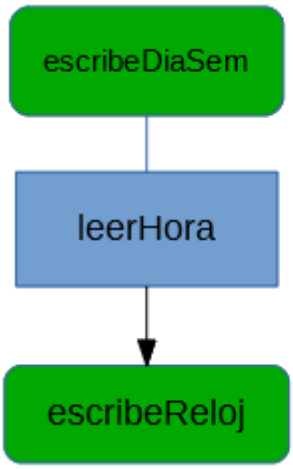



Figura 7.

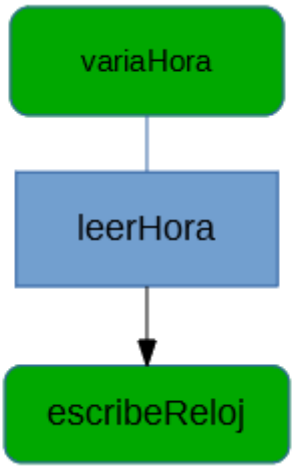



Figura 8.

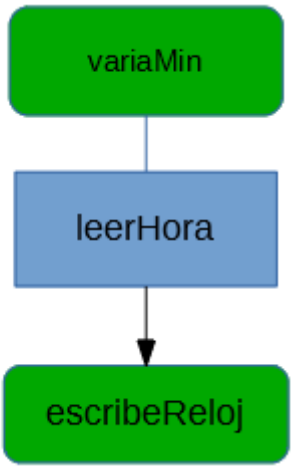



Figura 9.

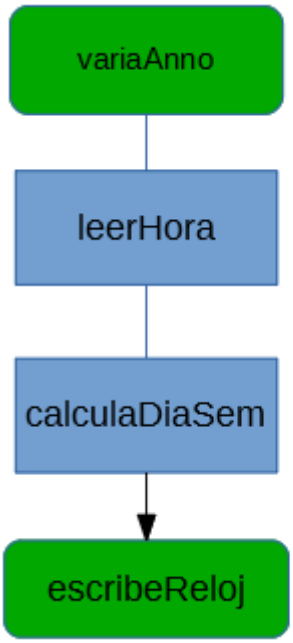



Figura 10.

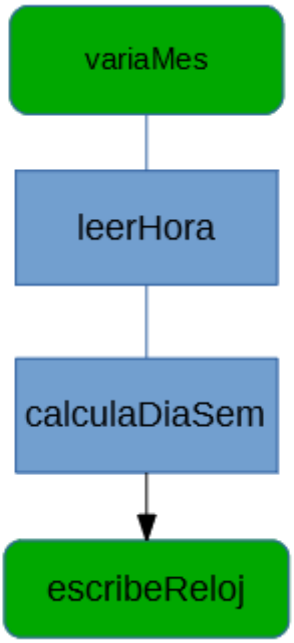



Figura 11.

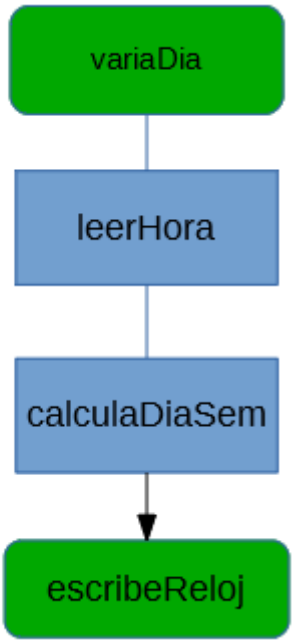



Figura 12.

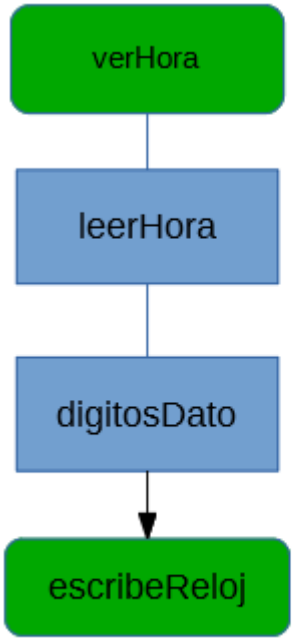



Figura 13.

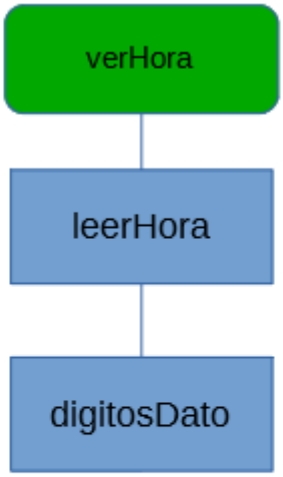



Figura 14.

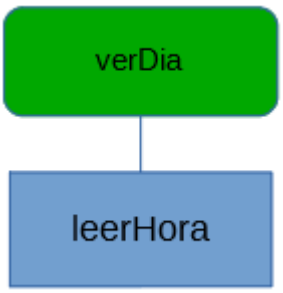



Figura 16.

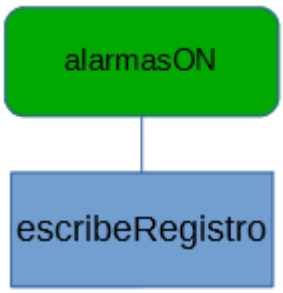



Figura 17.

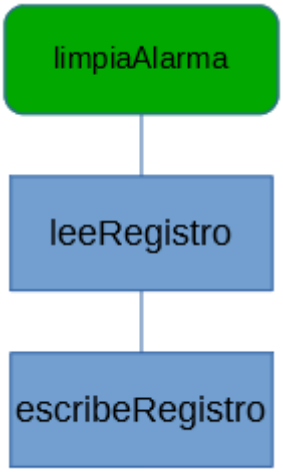



Figura 18.

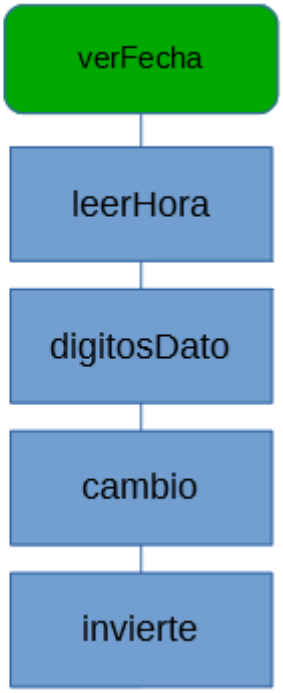



Figura 15.

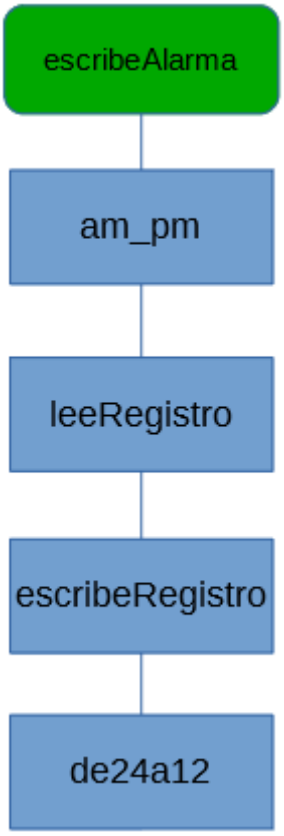



Figura 19.

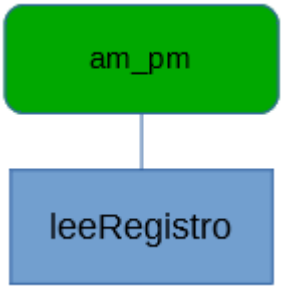



Figura 20.

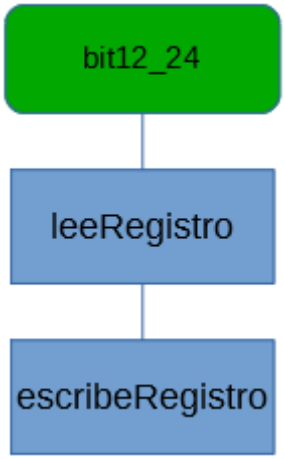



Figura 21.

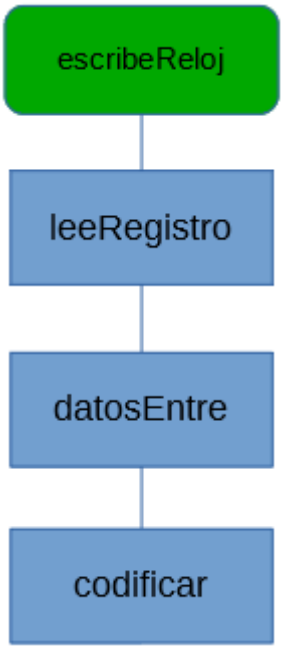



Figura 22.

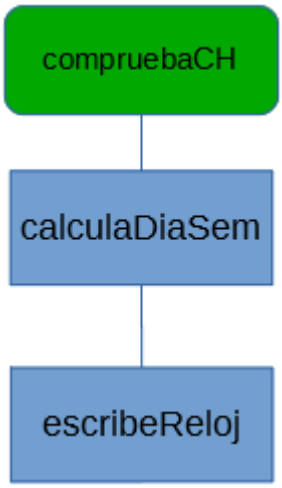



Figura 23.

Este esquema presenta todas las relaciones entre las funciones y ayuda a seguir mejor los diagramas de flujo de cada función, los cuales se presentan a partir de la siguiente página.

## 3.3- Diagramas de flujo de las funciones:

En este punto se muestran los diagramas de flujo que representan las funciones anteriormente mencionadas. Junto a la figura de cada función se resume brevemente la función.

*"leerHora"*:

Antes de almacenar el contenido de los registros se debe comprobar si se está en modo doce horas o veinticuatro, y en caso de ser modo doce horas, conocer el bit AM o PM. Después se puede establecer la comunicación con el reloj. Se irán leyendo los registros en orden y almacenando en una variable x con una estructura tipo Reloj. La lectura del registro de la hora depende del modo 12/24, para el modo doce horas se almacena el valor en *"x.hora12"* y *"x.hora"* se calcula a partir de ese valor. Para el otro modo se realiza el proceso contrario, se almacena *"x.hora"* y se calcula *"x.hora12"*. Una vez leídos todos los registros esta función manda la variable x a la función "compruebaCH" que comprueba si hay un cambio de hora y devuelve el valor de x.

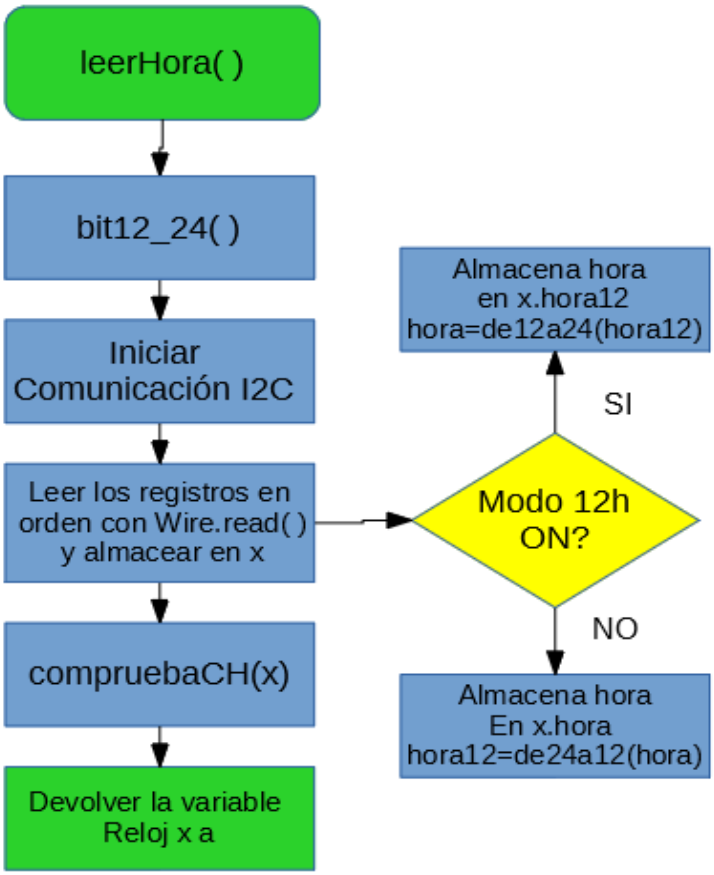



Figura 24.

***"Modo12h"*:**

Esta función se utilizará siempre que se quiera cambiar de modo, basta con mandarle ON (1 lógico) para establecer el modo doce horas u OFF (0 lógico) para el modo veinticuatro horas. Primero, la función necesita conocer el modo actual ("*AMPM*"), para así poder almacenar correctamente la hora a leer. Tras esto, se modifica la variable que almacena el modo y una variable auxiliar AP, que impedirá una nueva lectura que pueda cambiar "*AMPM*". Finalmente, la función manda la variable x para actualizar los registros.

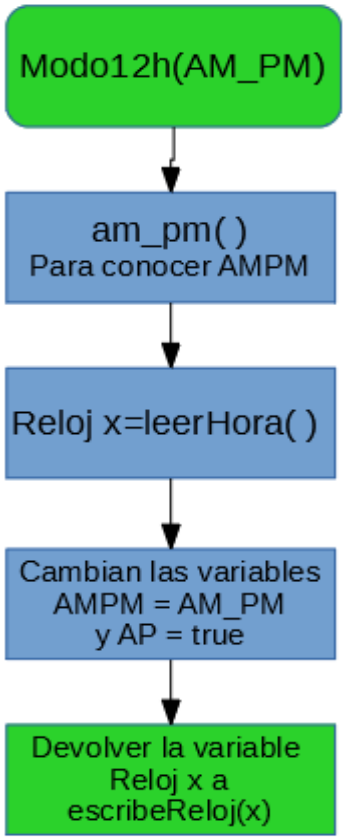



Figura 25

*"escribeHora"*, *"escribeFecha"* y *"escribeDia*Sem":

Estas funciones reciben los valores que se quieren escribir en los registros de tiempo, tras una lectura de los valores actuales de x, se modifican con los nuevos valores y se envían para ser escritos en los registros.

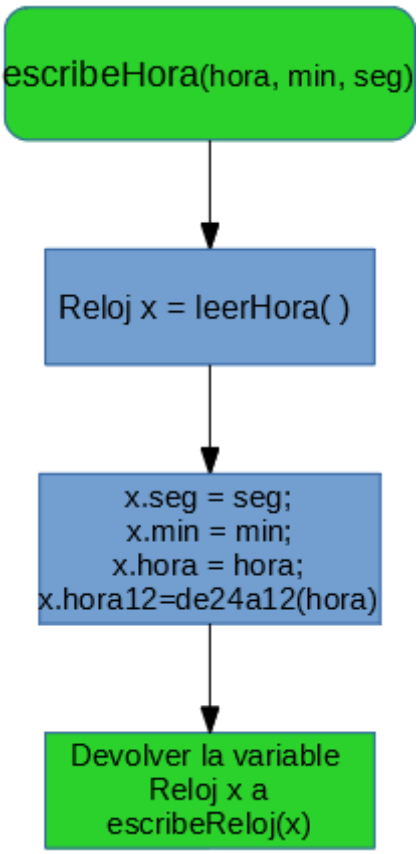



Figura 26.

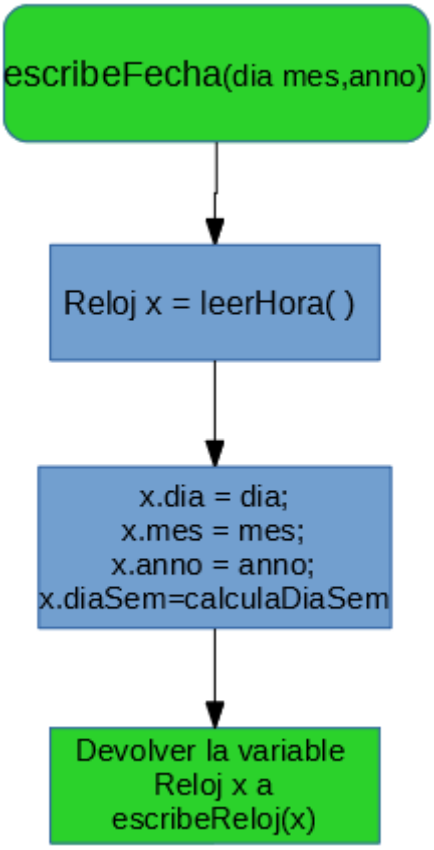



Figura 27.

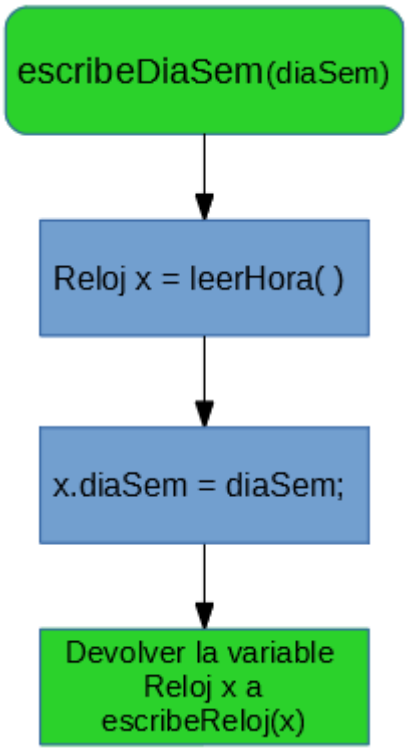



Figura 28.

***"variaHora"*:**

A esta función se le pasa una variable binaria para indicar si se quiere incrementar en una unidad la hora (1 lógico) o decrementar (0 lógico). En ambos casos se debe comprobar si se ha pasado el límite de la hora por encima o por debajo y modificar el valor si es necesario. Posteriormente se envía para su escritura en los registros.

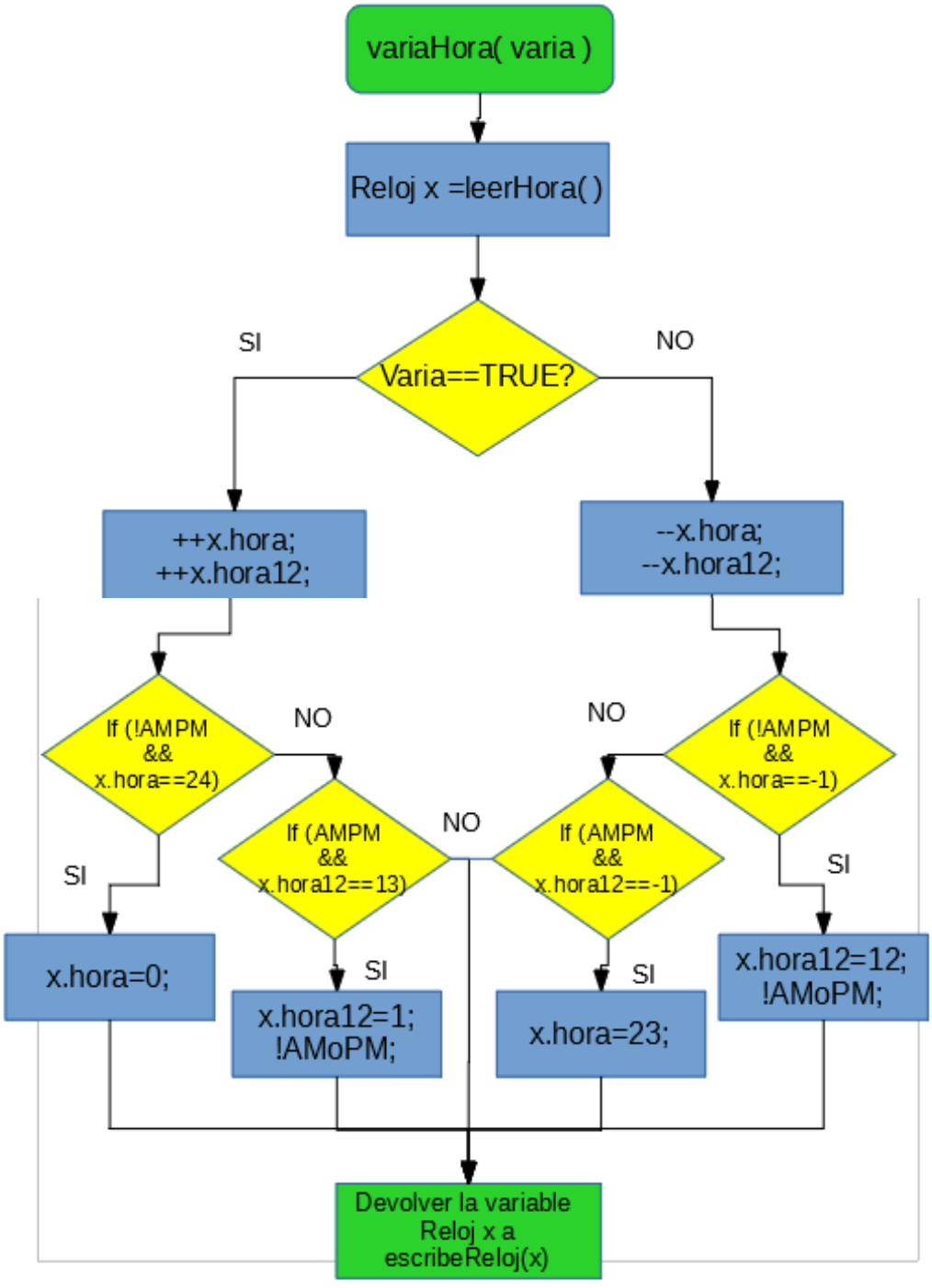



Figura 29.

***"variaMin"*:**

Esta función es como la anterior, pero para el caso de los minutos. No se muestra el diagrama del resto de funciones de variar, ya que la estructura es la misma, exceptuando que en esta se ponen los segundos a 0. En las demás lo único que cambia son los límites. Las funciones serían *"variaAnno"*, *"variaMes"*, *"variaDia"* y *"variaDia*Sem".

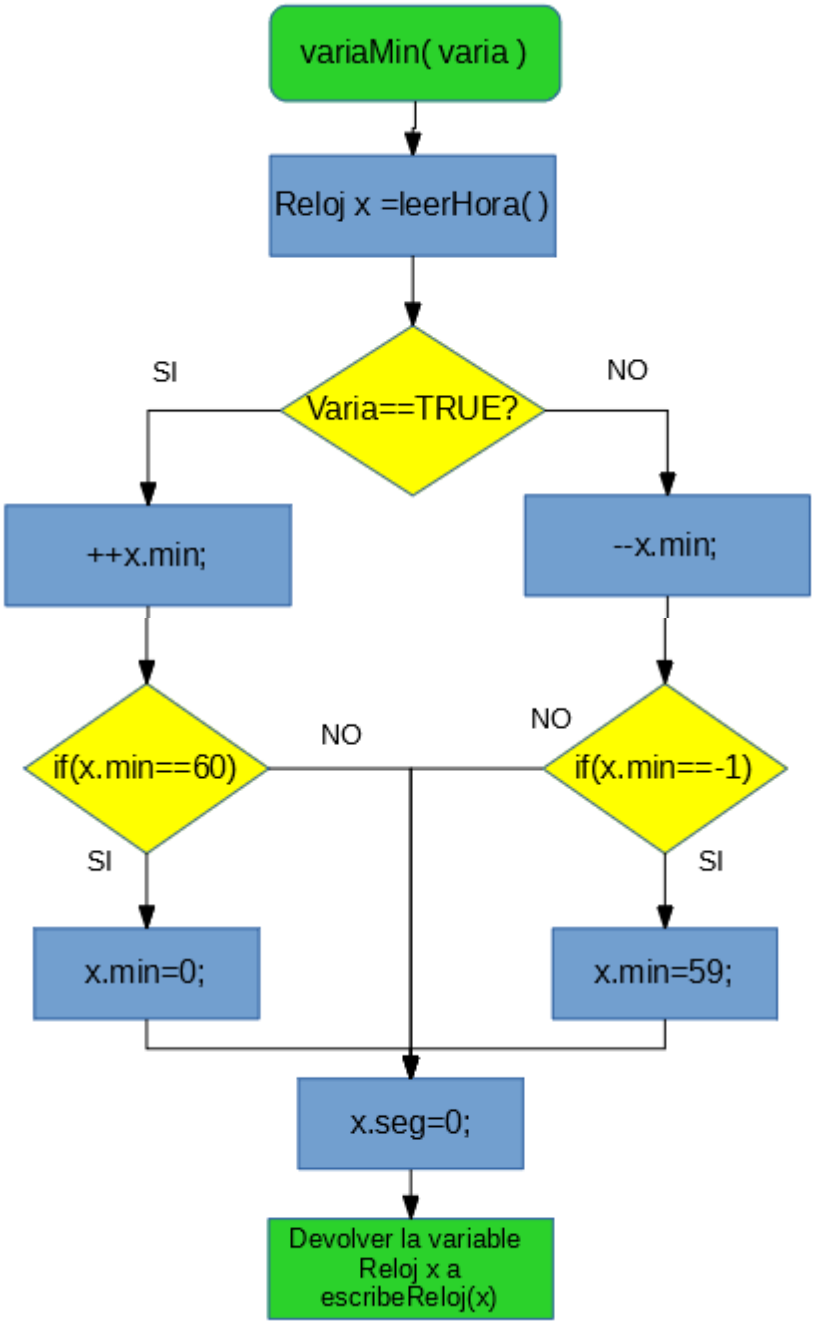



Figura 30.

***"verHora"*:**

Esta función pasa el valor de los registros de la hora a una cadena. Recibe un parámetro llamado formato, que puede ser largo o corto, dependiendo si se quiere añadir los segundos o no. Puntualizar que cuando se dice comprobar se refiere a una operación condicional con el valor binario que devuelve *"digitosDato"*, para saber si es necesario añadir un 0 o no. Añadir se refiere a sumar el carácter a la cadena "salida".

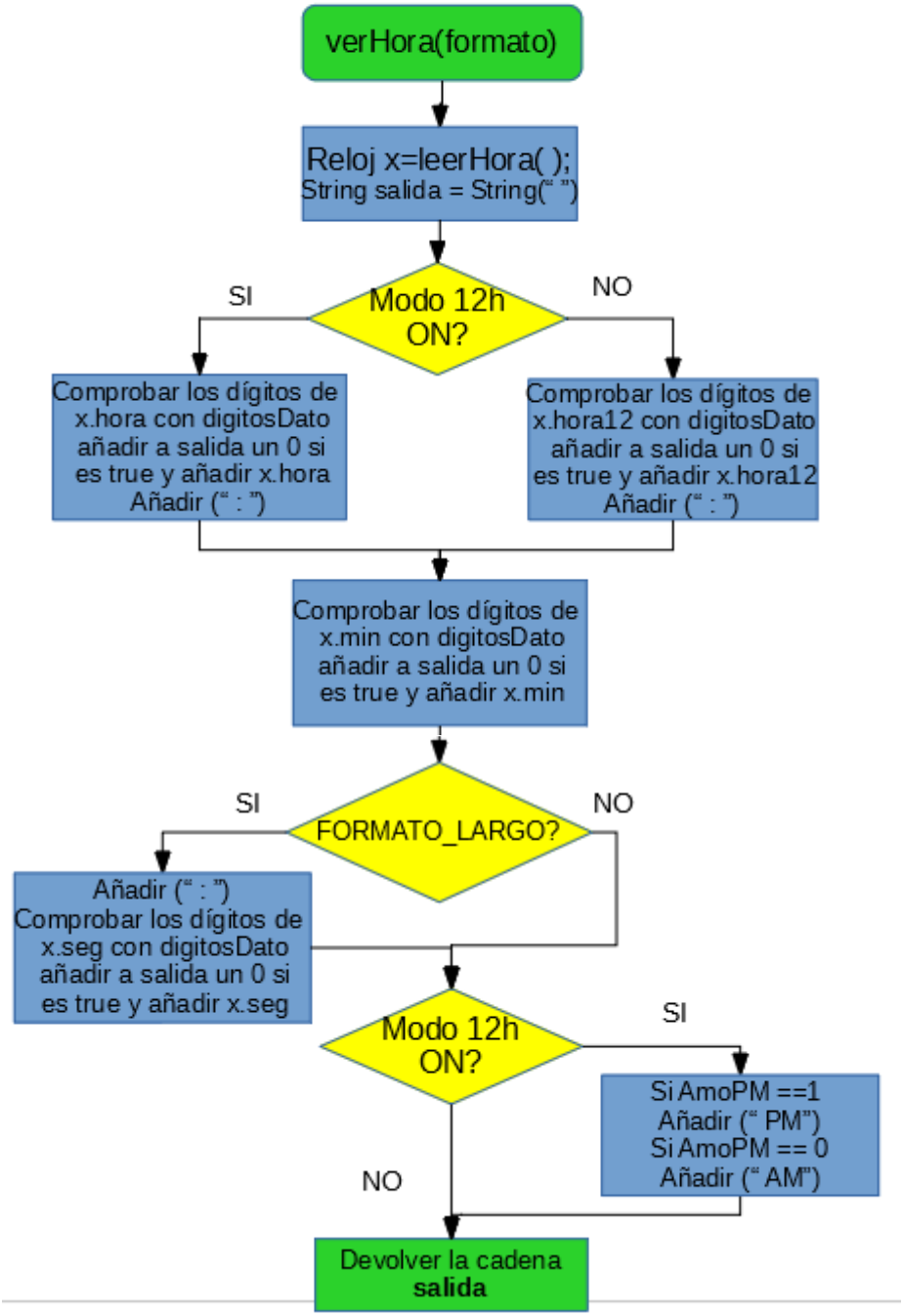



Figura 31.

*"verFecha"*:

Como el caso anterior, esta función convierte los datos almacenados en una cadena de caracteres, en este caso los valores de la fecha. También puede tener formato corto o largo, mostrando el formato largo el año completo y el formato corto solo la década. Por defecto la cadena de la fecha será día/mes/año, pero se puede elegir un segundo orden, año/mes/día, o un tercer formato, mes/día/año.

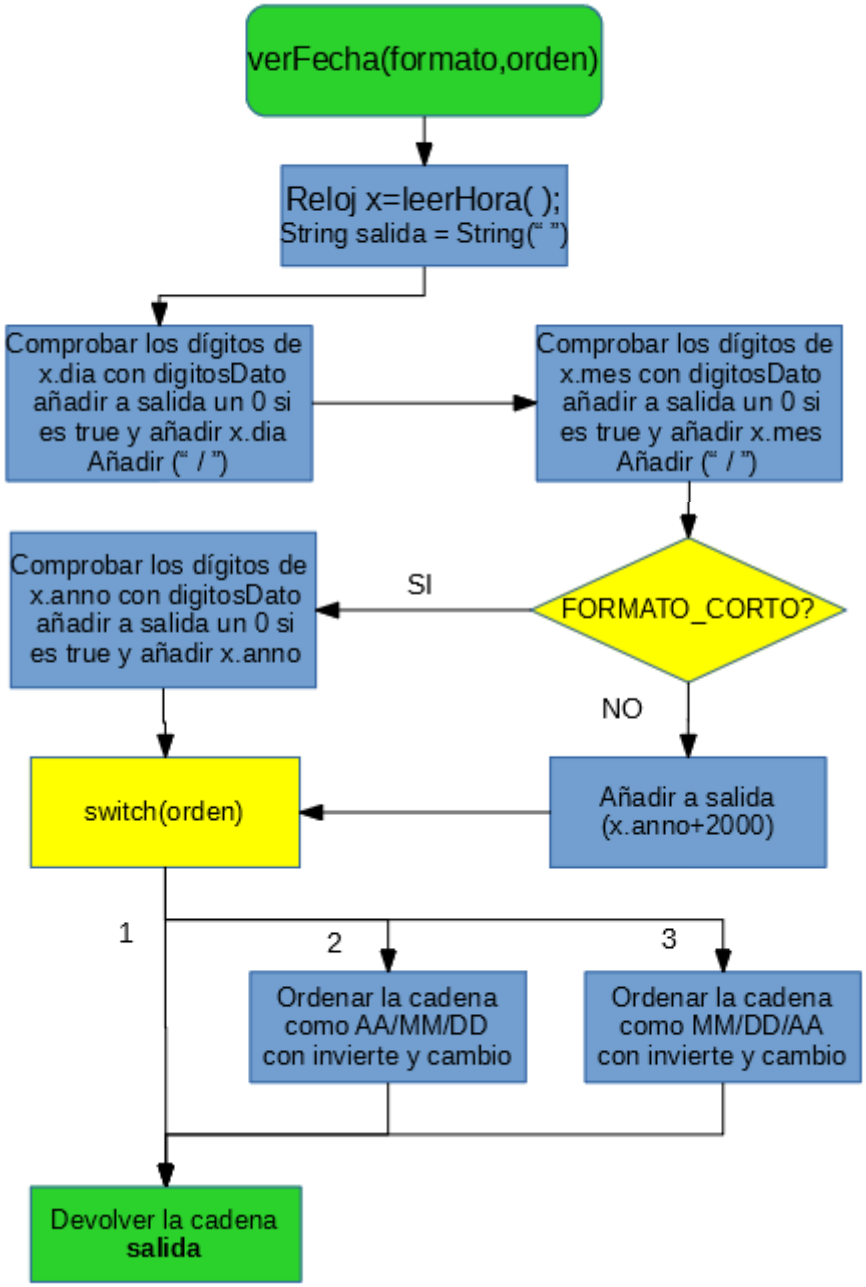



Figura 32.

***"verDia"*:**

Función que pasa a una cadena el día de la semana. Mediante un case se elige el día y en caso de que el formato elegido sea corto, se eliminan los caracteres desde la tercera posición. No se muestra el diagrama de *"verMes"*, ya que es la misma estructura que la de *"verDia"*, pero con los doce meses en vez de los días de la semana.

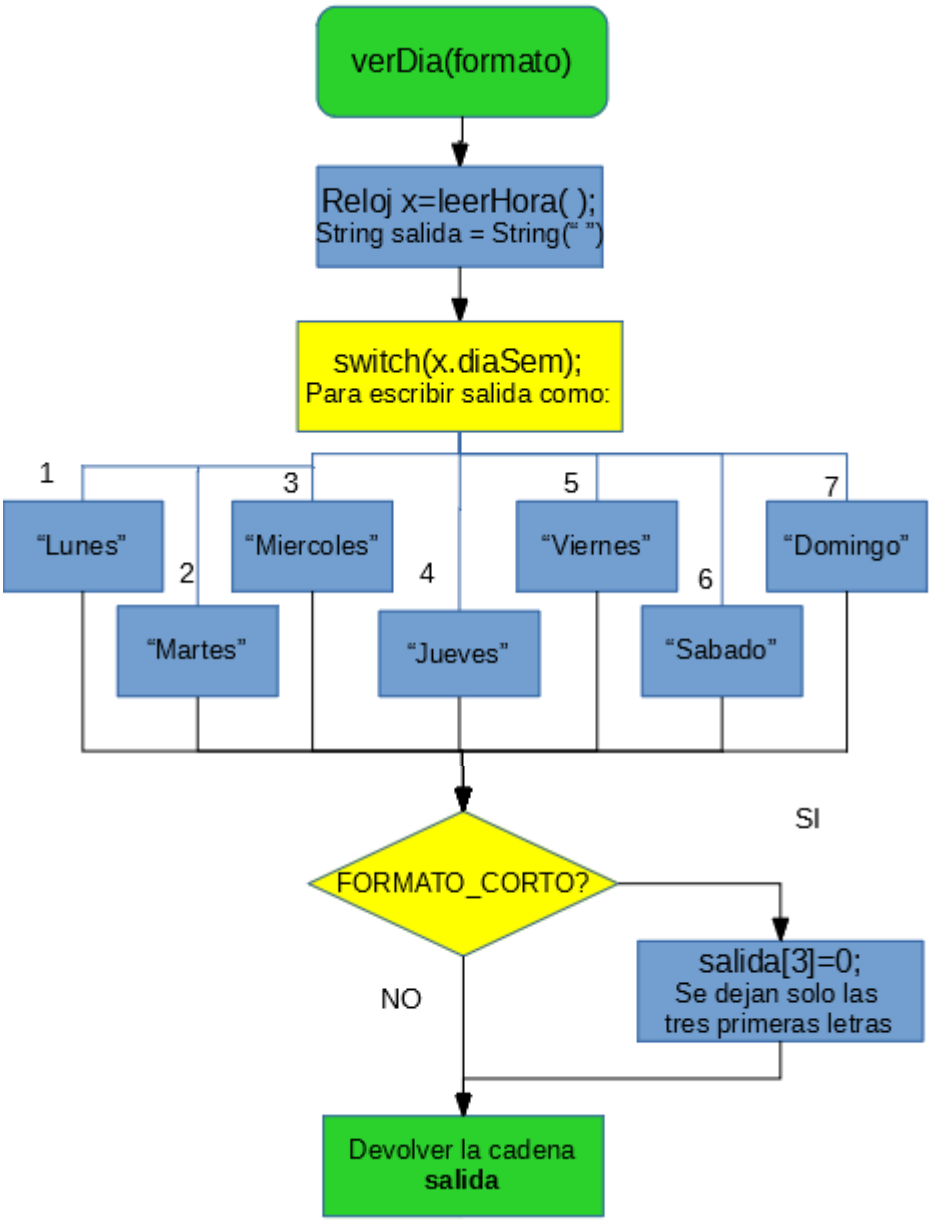



Figura 33.

***"alarmasON"*:**

Para activar las alarmas se pasará el parámetro tipo, que indica la alarma que se quiere establecer. Se deben poner a 1 los bits del registro de control que activan la alarma, después se comprueba el tipo elegido con una sentencia switch. Se han definido ocho de las alarmas posibles, cuatro para cada alarma, que se establecen en función del bit más significativo de los registros de alarmas. El tipo de alarma se almacena en una variable global llamada "*tipoAl*", para que la función que escribe las alarmas sepa que tipo es.

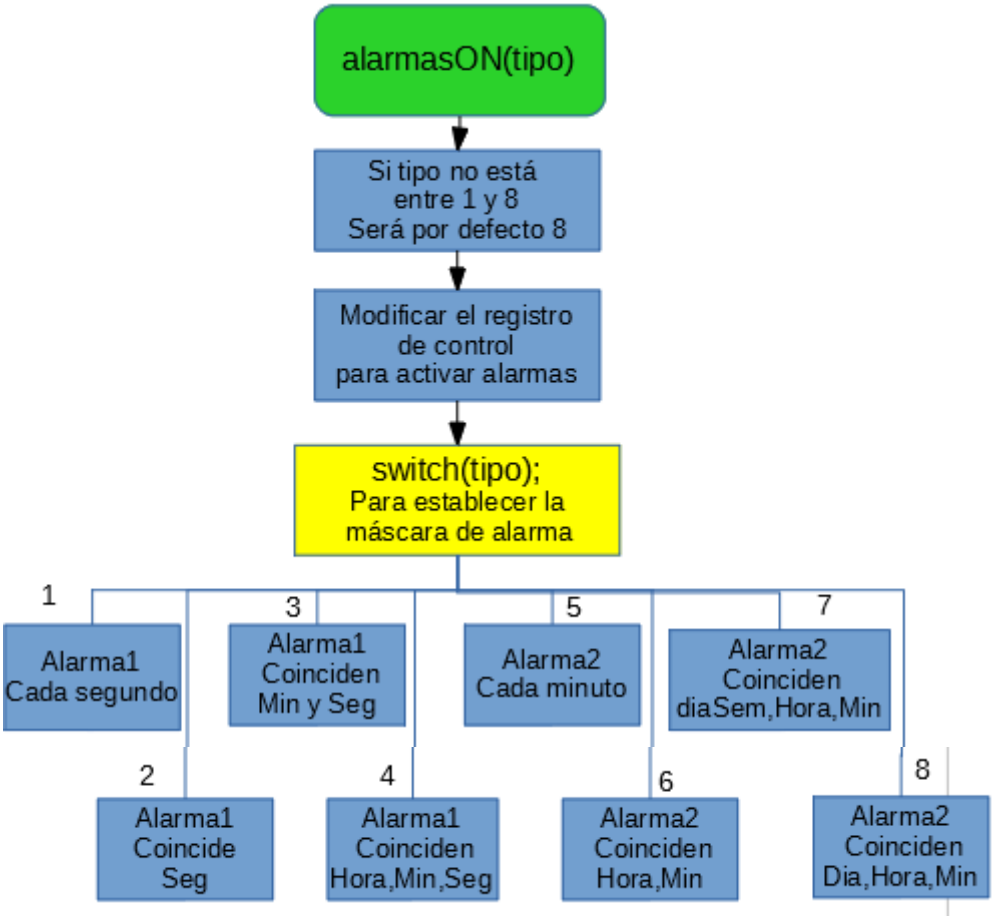



Figura 34.

*"escribeAlarma"*:

Esta función se utilizará tras haber activado las alarmas con la función anterior. Las variables que recibe son los datos para establecer la alarma en día, hora, minutos y segundo. Si no se va a utilizar alguno de los datos no importa el valor que tenga. Los cuatro primeros tipos de alarma trabajan con la alarma 1 y los otros cuatro con la alarma 2. En ambas alarmas se opera igual, escribiendo en los registros los valores codificados, para el registro de la hora, como en otras funciones, debe comprobarse en que modo está y modificar el valor de la hora si fuera necesario.

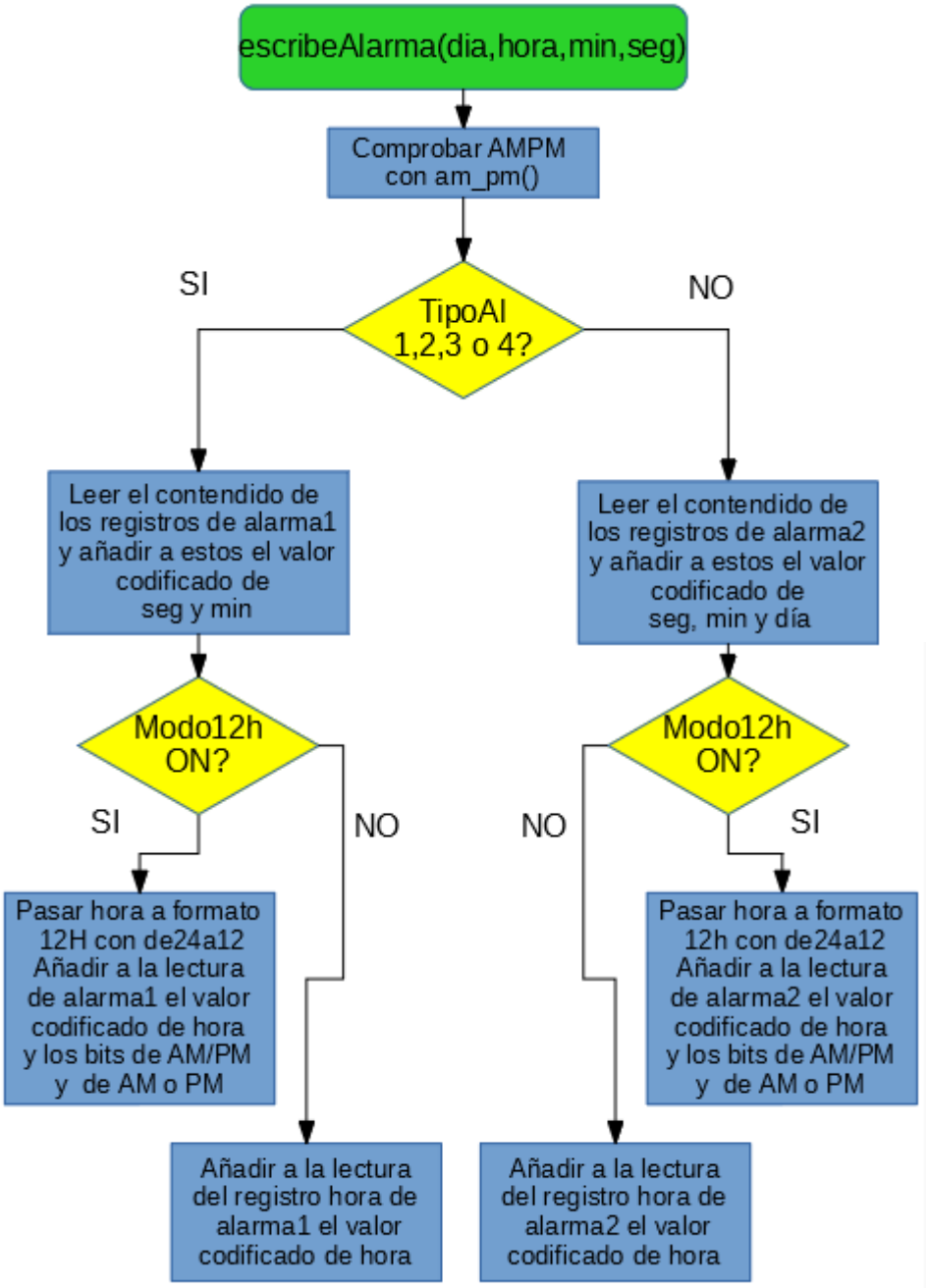



Figura 35.

*"limpiaAlarma"*:

Función a utilizar tras la activación de cualquier alarma, para reestablecer el registro de estado.

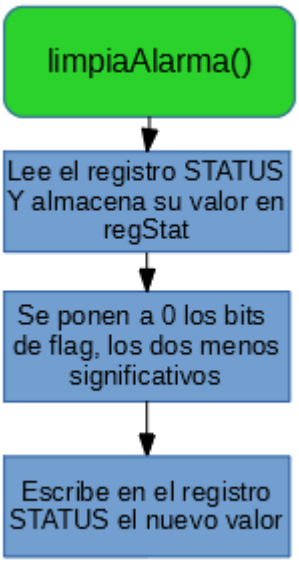



Figura 36.

**Funciones Entre:**

Funciones para comprobar si los valores de los datos de tiempo son válidos. Faltan los diagramas de *"minsegEntre"*, *"diaEntre"*, *"diaSemEntre"*, *"mesEntre"* y *"annoEntre"*, cuyas estructuras son iguales que las siguientes.

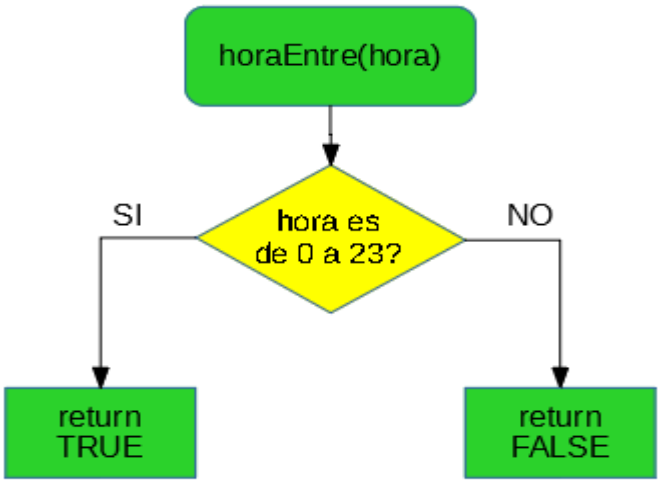



Figura 37.

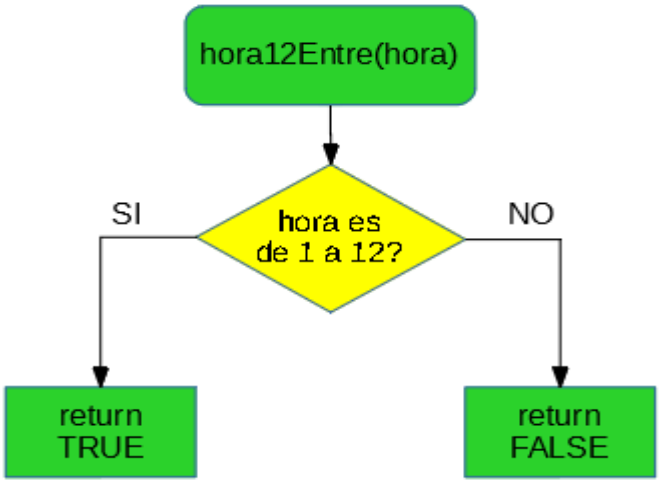



Figura 38.

**“*cambio*” e “*invierte*”:**

Estas dos funciones, como ya se ha mencionado anteriormente, en la función “verFecha”, sirven para reordenar la cadena de caracteres de la fecha. Como se observa en los dos diagramas, estas funciones devuelven una variable auxiliar, que es la cadena de caracteres reordenada. La función cambio recibe dos posiciones y la cadena completa, para realizar el intercambio de los caracteres de dichas posiciones. En el caso de la función invierte, solo recibe la cadena de caracteres y sustituye todas las posiciones simétricamente, las cinco primeras por las cinco últimas, tiene siempre diez caracteres, ya que solo se usa al reordenar el formato largo de día, mes y año.

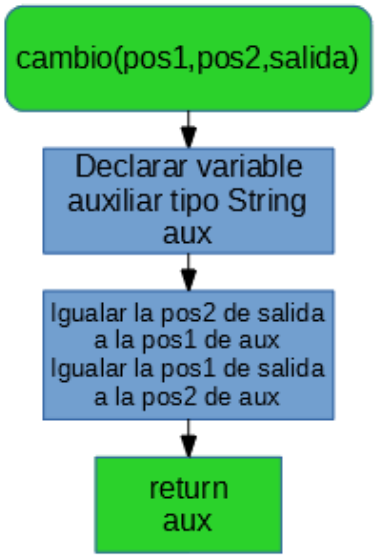



Figura 39.

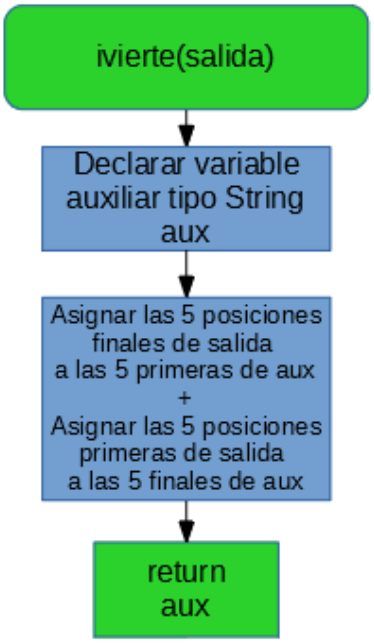



Figura 40.

***”digitosDato”*:**

Una simple función booleana, que devuelve true si el dato enviado es de un dígito.

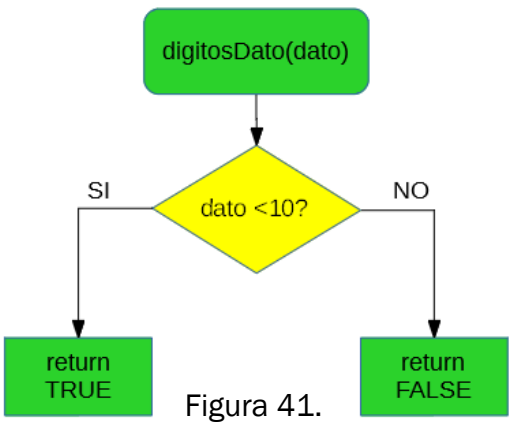



Figura 41.

***"escribeReloj"*:**

Esta es una de las funciones más importantes, la que escribe en los registros. Primero comprueba si la variable global AP es *true* o *false*, ya que puede haberla cambiado a *true* la función *"Modo12h"* y no se almacenarían los datos actuales. Después se va escribiendo en orden cada valor en su registro, previa confirmación de que el valor es correcto con las funciones "Entre", si no es válido se escribe el dato actual. Como en otras funciones, el registro de la hora tiene la excepción de que se debe comprobar en qué modo está.

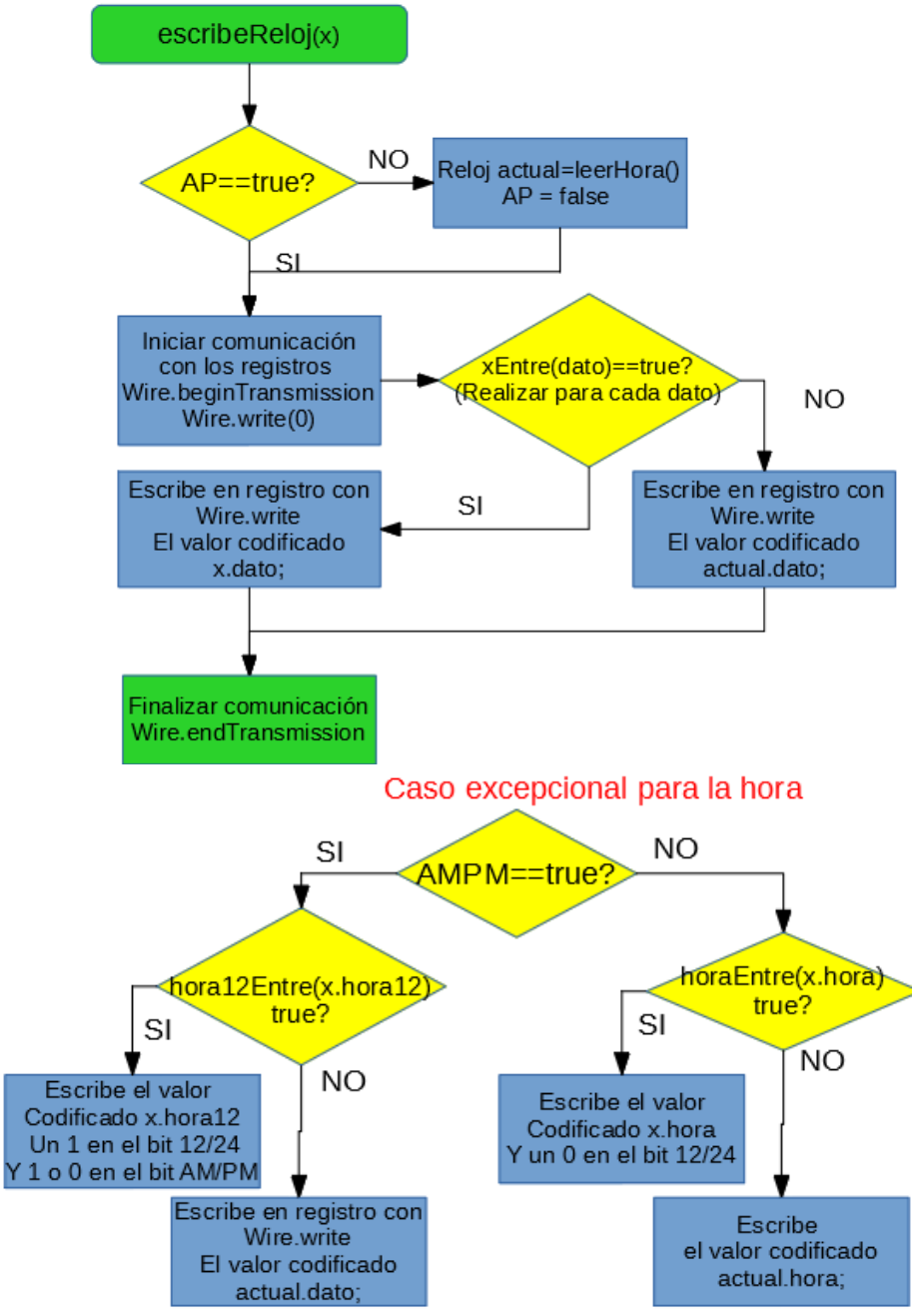



Figura 42.

**“*decodificar*” y “*codificar*”:**

A la hora de leer y escribir en los registros necesitamos decodificar y codificar los datos de la variable x respectivamente.

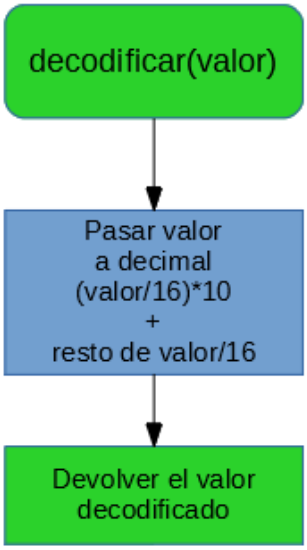



Figura 43.

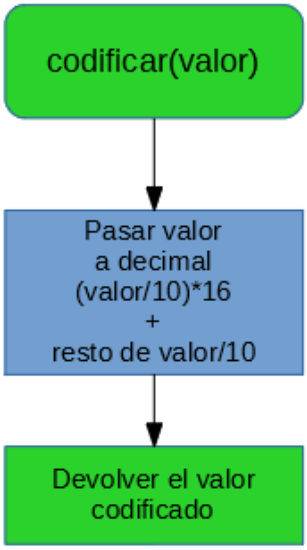



Figura 44.

***“leeRegistro”* y *“escribeRegistro”*:**

Como sus nombres indican, la función *“leeRegistro”* lee el byte completo de la dirección que se le pasa y la función *“escribeRegistro”* sustituye el contenido que tiene el registro de la dirección indicada por el valor solicitado.


Figura 45.


Figura 46.

*"am_pm"*:

Función importante para el registro de la hora, ya que es la que lee y almacena el estado del bit que cambia de modo.

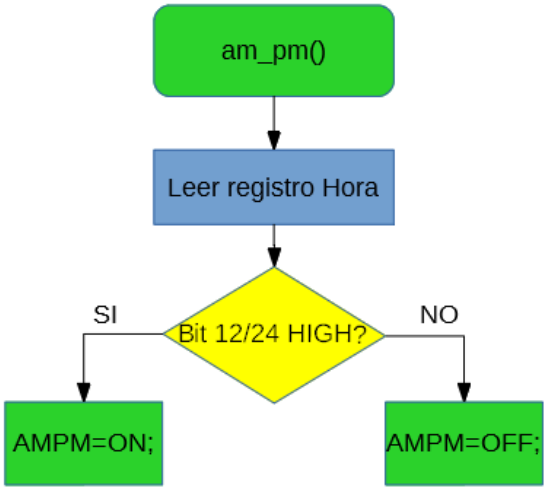



Figura 47.

*"bit12_24"*:

Como la anterior, es importante para el registro de las horas, en esta ocasión para asegurar que está en el modo horario correcto y conocer el bit AM o PM.

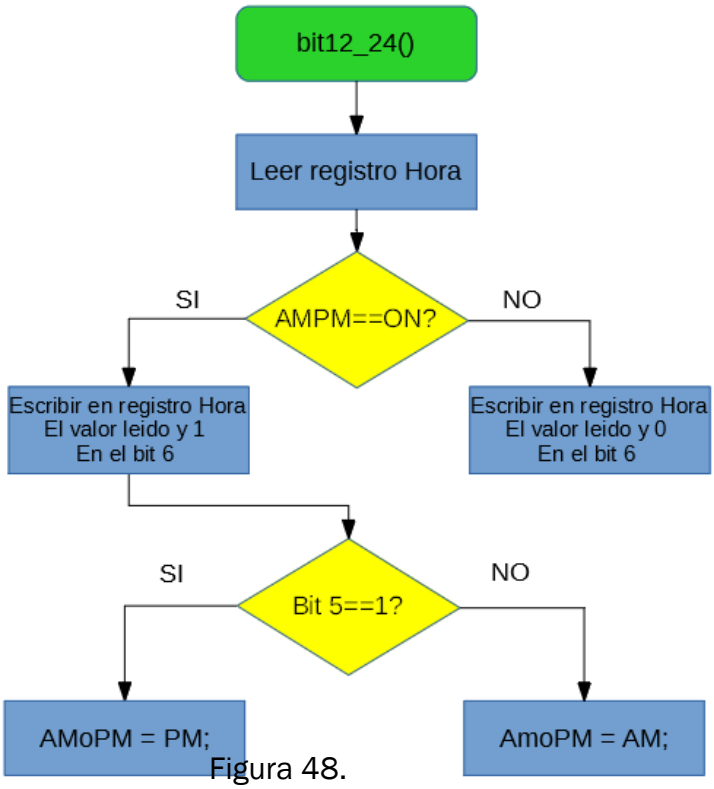



Figura 48.

*"de12a24"* y *"de24a12"*:

Como las dos anteriores, también serán necesarias para el cambio de modo. Reciben la hora en un modo y la devuelven en el otro.

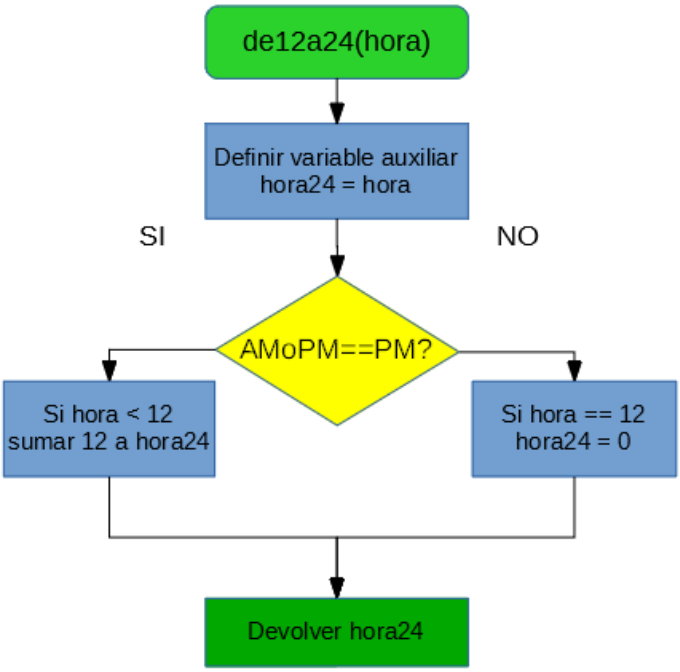



Figura 49.

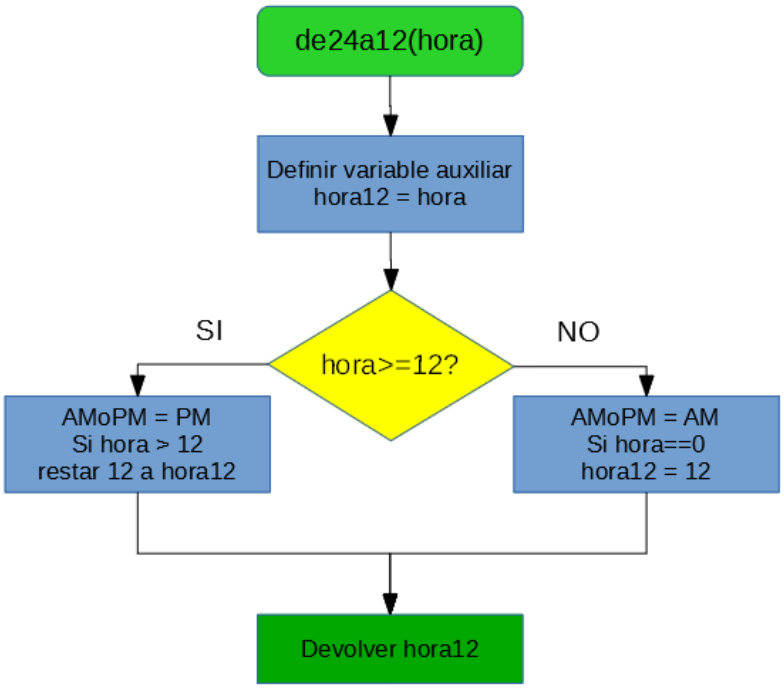



Figura 50.

*"compruebaCH"*:

Al realizar una lectura de los registros se comprueba si debe realizarse un ajuste de hora. Para realizar este ajuste son necesarias tres variables, "*horaMas*", será el próximo día en el que se debe adelantar la hora, "*horaMenos*", el día en que se debe retrasar y cH, una variable binaria que indica si el próximo cambio es "*horaMas*" (cH=*false*) o si es "*horaMenos*" (cH=*true*). En caso de que haya una coincidencia de fecha con "*horaMas*" u "*horaMenos*", se modifica la variable "*x.hora*" y se escribe en el registro, tras esto es necesario calcular el cambio de hora del siguiente año. La función devuelve la variable x, se haya modificado o no el valor de la hora.

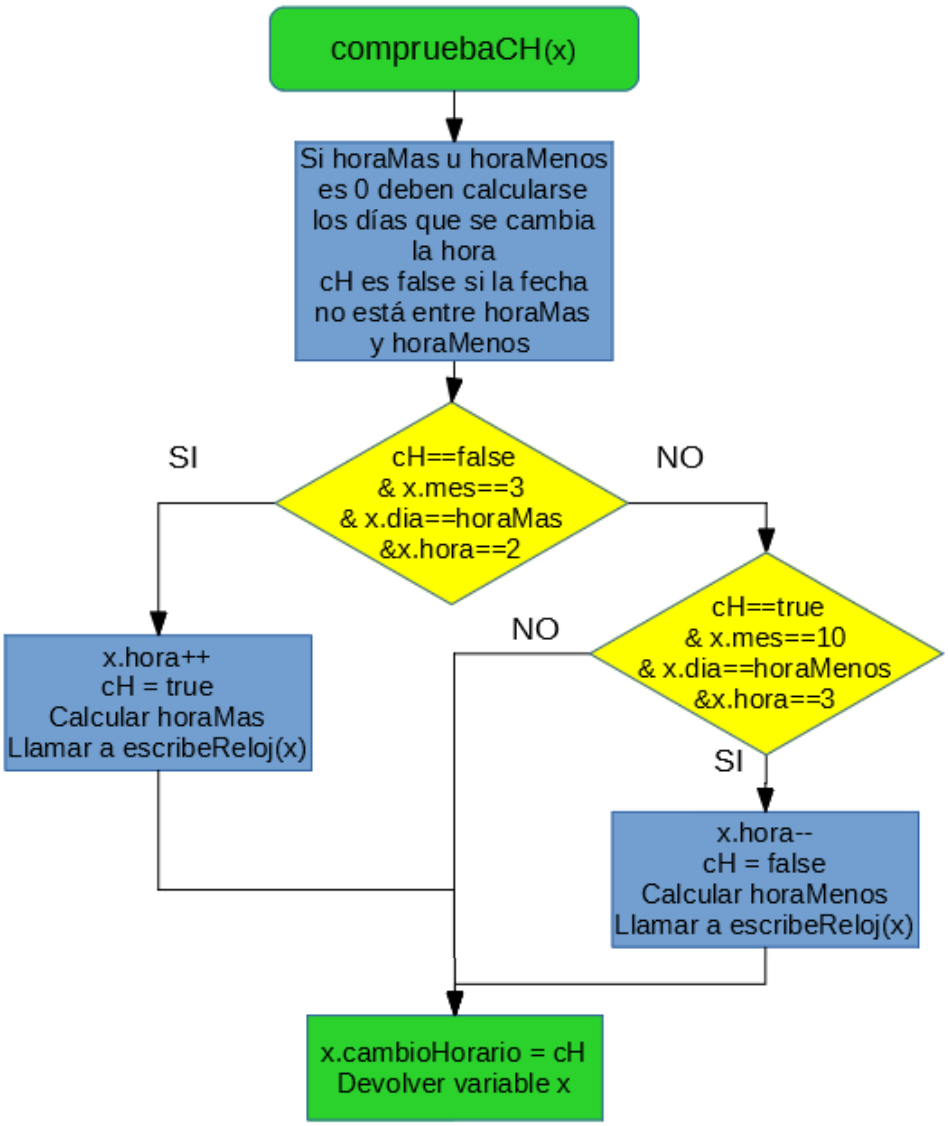



Figura 51.

*"calculaDiaSem"*:

Esta última función recibe el día, mes y año, y mediante un algoritmo, llamado congruencia de Zeller, que obtiene el día de la semana. Siendo el resultado un número del cero al seis, donde el cero es el domingo y el resto van en orden de lunes a sábado. Para no aumentar el código en las funciones más importantes como la de leer, se ajustará a la definición dada a los días de la semana, donde el domingo no es cero si no 7.

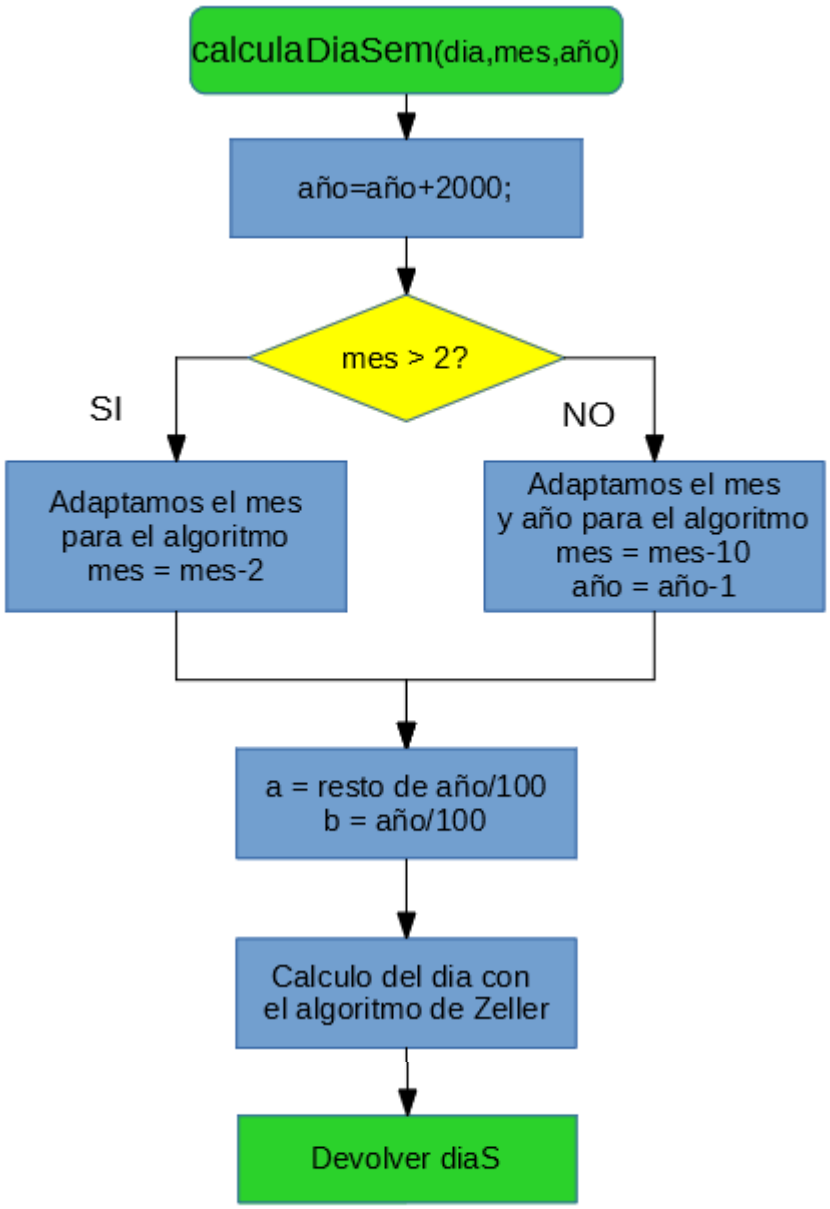



Figura 52.

# 4- Implementación de la librería

## 4.1- Desarrollo de las funciones:

Aunque en el código del programa las funciones y variables declaradas son auto-explicativas y se añaden suficientes comentarios para la compresión de la librería, en este apartado se realiza una descripción más detallada de cada función. Además, en el Anexo 4 se presenta el manual de usuario con el fin de que no haya dudas a la hora de utilizar la librería.

Las descripciones se presentan en el mismo orden que los diagramas anteriores, que es el orden en que aparecen en el código.

### Clase Reloj:

**-Reloj:**

En esta clase se crea la estructura Reloj, la cual es pública y contendrá los valores de día, mes, año, día de la semana, hora, hora modo 12 horas, minutos, segundos y un último que no pertenece a los registros del RTC, la variable "cambioHorario", la cual se modifica en la función *"compruebaCH", e indica si es horario de verano o de invierno*.

Todos los valores son variables uint8_t, ya que solo necesitan un byte, a excepción del año que es uint16_t y "cambioHorario" que es binaria.

En el resto de la librería, esta estructura se utilizará con la variable x.

### Clase RTC:

***-Funciones públicas:***

Aquí se muestran las funciones de la interfaz, que se podrán llamar desde el programa principal cuando se haya añadido la librería al código fuente.

**"RTC":**

El fin de esta función es inicializar la clase para asignarle un nombre, en los programas de ejemplo se utiliza rtc. Además, esta función llama a la función begin de la librería wire, la cual inicia la conexión con el bus $I^2C$.

***"leerHora"*:**

Como ya se ha mencionado en este trabajo, esta es una de las funciones más importantes de la librería, siendo llamada en mayoría de las funciones públicas.

Lo primero que hace esta función es llamar a la función "bit12_24", para comprobar en que modo está trabajando el reloj.

Tras esto se prepara la comunicación con el bus $I^2C$ mediante las siguientes funciones de la librería Wire:

- *beginTransmission*, para iniciar la transmisión al dispositivo esclavo, cuyo argumento es la dirección del mismo, en este caso la dirección del RTC, su valor es 0x68.
- *write* o *send* (depende de la versión de Arduino), estas se utilizan para poner en cola los bytes de la comunicación, como queremos que lea todos los registros de tiempo se le pasa como argumento el valor 0, para que comience en el principio.
- *endTransmission*, indica que se puede comenzar la transmisión, liberando el bus cuando la comunicación ha terminado.
- *requestFrom*, esta función sirve para solicitar bytes de un esclavo. Los argumentos que se le pasan son la dirección del dispositivo, en este caso la del RTC, y la cantidad de bytes que se quieren leer del mismo. Como esta función lee todos los registros de tiempo, se le pasa la dirección 0x68 y se solicitan siete bytes.
- *read* o *receive* (depende de la versión de Arduino), lee un byte transmitido con las funciones anteriores.

Una vez establecida la conexión mediante las primeras cuatro funciones explicada arriba, se define una variable x tipo Reloj, para almacenar los valores de los registros.

Para almacenar los valores, se va asignando a cada componente de la estructura un valor que se obtiene al decodificar el byte leído mediante la función *read*, al que se le aplica una máscara AND para limpiar los bits innecesarios. Esto se debe realizar en el orden que tienen los registros y que se mostró en la Tabla 1, segundos, minutos, hora, día de la semana, día, mes y año.

Las máscaras utilizadas son "01111111" para los segundos y minutos. En el caso de las horas depende del modo, para el modo de 24h se ponen a cero los dos bits más significativos "00111111" y para el modo 12h debe ignorarse el bit 5 a la hora de almacenar el valor, ya que será 1 si la hora es PM, "00011111". El byte de día de la semana solo

necesita tres bits, ya que va de uno a siete, "00000111". De los meses solo interesa hasta el quinto bit como en el modo 12h de la hora. Por último, el día puede prescindir del bit seis y siete, por lo que usa la misma máscara que la hora en formato 24h.

En el momento de almacenar la hora, mediante un condicional, se comprueba que modo está activo. En caso de ser modo 24h, se almacena el registro en *"x.hora"* con la máscara antes indicada, y el valor de *"x.hora12"* se obtiene mediante la función "*de24a12*". Si el modo es 12h, se realiza el proceso contrario, se almacena *"x.hora12"* y se calcula "*x.hora"* con la función "*de12a24*".

Para finalizar con esta función, se manda la variable x a la función que comprueba si hace falta un cambio de hora.

La función *"leerHora"* devuelve la estructura x.

### *"Modo12h"*:

El objetivo de esta función es poder cambiar de modo de trabajo al reloj durante el funcionamiento del mismo. Para ello, el argumento que solicita la función es una variable booleana, la cual indica si el modo es 12h cuando es un uno lógico y modo 24h si es un cero lógico.

En primer lugar, la función llama a *"am_pm"*, que consulta el estado actual del bit que indica el modo y lo almacena en "*AMPM*". Esto es necesario, ya que se va a realizar una lectura y si el valor de "*AMPM*" no es el correcto, por ejemplo, al iniciar el programa, que por defecto es *false*, podrían almacenarse valores de hora erróneos.

Se leen los valores de los registros mediante *"leerHora"* y se almacenan en una variable x.

La variable global "*AMPM*" se actualiza con el parámetro recibido por la función *"Modo12h"* y otra variable global llamada AP se pone a *true*, para que no se realice una nueva lectura que pueda variar el valor de la variable "*AMPM*" en la función *"escribeReloj"*.

Se escriben los datos de x mediante la función *"escribeReloj"*. En este caso como la variable "*AMPM*" ya se ha modificado, se escribirá "*x.hora"* si se ha seleccionado el modo 24h y *"x.hora12"* si el modo es 12h.

La función no devuelve ningún valor.

*"escribeHora"*:

Función con la que se puede establecer la hora deseada en el RTC. Para ello es necesario añadir los argumentos de hora, minutos y segundos. Si alguno de los parámetros no se añade, el valor por defecto será 0.

Se lee el reloj y se almacenan los valores en x.

Los valores de hora, minutos y segundos se sustituyen por los recibidos por la función *"escribeHora"*.

Como en la función anterior, se pone la variable AP a true. En esta ocasión es debido a que la función "escribeHora" recibe el parámetro de la hora en formato 24h, calculando *"x.hora12"* mediante la función "*de24a12*", por lo que al llamar a la función *"escribeReloj"*, la cual llama a *"leerHora"*, puede ser modificado el valor de la variable global "*AMoPM*".

La función finaliza sin devolver ningún valor, al llamar a *"escribeReloj"*, a la que se le pasan los nuevos valores de x.

*"escribeFecha"*:

Es como la función anterior, pero los parámetros que recibe son el día, el mes y el año.

Lo único distinto a tener en cuenta es que el año se almacena solo desde la década, por lo que cuando el parámetro que se quiere escribir es mayor de 99, y dado que el reloj trabajará de 2000 a 2100, se ajusta el año restando 2000 al parámetro recibido.

Por último, antes de llamar a *"escribeReloj"*, se almacena también el día de la semana en *"x.diaSem"* mediante la función *"calculaDiaSem"*.

"escibeDiaSem":

Es igual que las de fecha y hora, pero solo para el día de la semana.

Aunque la función *"escribeFecha"* también escribe el día de la semana, se ha creado esta función por si se desajustara el reloj y se quisiera modificar solamente el día de la semana.

***"variaHora"*:**

Esta función, al igual que el resto de las funciones "varía", reciben un parámetro binario, el cual será un 1 lógico si se quiere aumentar el valor del parámetro asociado a cada función y un 0 lógico si se quiere decrementar.

Deben almacenarse en x los valores actuales del reloj.

Si el parámetro recibido es true, se incrementa *"x.hora"* y *"x.hora12"*.

En caso de que el nuevo valor sea 24 para el modo 24h o 13 para el modo 12h, se debe poner *"x.hora"* a 0 o *"x.hora12"* a 1 y "*AMoPM*" al valor contrario al actual respectivamente.

Si el parámetro es false, se decrementa *"x.hora"* y *"x.hora12"*.

Como antes se debe comprobar el nuevo valor, ahora si es -1 se cambiará a 23 para el modo 24h y a 12, cambiando el valor de "*AMoPM*", para el modo 12h.

Los nuevos valores de x se pasan a *"escribeReloj"* para sustituirlos en el RTC.

***"variaMin"*:**

Si el parámetro "varia" es un 1 lógico se aumentan los minutos y en caso de llegar a 60 se cambia el valor a 0.

Cuando sea un 0 lógico, se decrementa "x.min" y si el valor nuevo es -1 se modifica a 59.

Tanto si se incrementa como si se decrementa, se inicializarán los segundos a 0.

***"variaAnno"*:**

Siguiendo con esta serie de funciones varia, esta tiene la misma estructura que la anterior, pero en caso de que el año llegue a 100 pasará a ser 0 y si llega a -1 pasa a ser 99.

Además, se modificará también el día de la semana llamando a *"calculaDiaSem"*.

***"variaMes"*:**

La diferencia con *"variaAnno"* es que si "*x.mes"* alcanza 13 incrementando, el valor pasa a ser 1, y si llega a 0 decrementando, pasará a ser 12.

***"variaDia*Sem":**

Debe compararse el nuevo valor a 8 si se incrementa, en caso de ser verdadero se modifica y *"x.diaSem"* pasa a ser 1, si se decrementa se compara a 0 y si se cumple, se modifica "x.*diaSem*" a 7.

***"variaDia":***

Por último, la función que cambia el día, para terminar con esta serie de funciones que modifican los registros de tiempo en una unidad.

Funciona como las anteriores, pero en este caso la comparación es más compleja, ya que depende del mes.

Si el mes es abril, junio, septiembre o noviembre, al llegar a 31 cuando se incrementa, se vuelve al día 1, si se llega a 0 al decrementar, el día será el 30.

Para febrero, si se alcanza el día 30 incrementado, pasa a ser 1, y si se llega a 0 al decrementar, pasa a ser 29.

El resto de los meses el límite ascendente es si llega a 32 y si descendiendo llega a 0, se modifica a 31.

***"verHora":***

Las funciones "ver" se encargan de convertir los datos de los registros, que han sido decodificados y almacenados por la función *"leerHora"*, a una cadena de caracteres. Dichas cadenas serán distintas dependiendo de los datos, el formato y en el caso de la fecha, el orden.

Todas las funciones "ver" devuelven una cadena y a la hora de formarla acuden a la función *"digitosDato"*, a la cual se le pasa como argumento el dato que se va a añadir a la cadena, y esta comprueba si es de uno o dos dígitos, en caso de ser de un dígito la función devuelve true y si es de dos dígitos devuelve false.

Los datos se leen de los registros y se almacenan en la variable x.

Conocidas las coincidencias de las estas funciones, se explican a continuación las diferencias para la función *"verHora"*.

Se debe comprobar si es modo 12h, "*AMPM*" es un 1 lógico, o si es modo 24h, "*AMPM*" es un 0 lógico. En ambos casos se añade la condición con la función *"digitosDato"*, añadiendo un 0 a la cadena en caso de que se cumpla. Cuando el modo sea 12h se añade *"x.hora12"* y cuando sea modo 24h se añade "*x.hora"*. Seguido a la hora se añade el carácter ":". Una vez se tiene la hora, se hace el mismo proceso de comprobar los dígitos para los minutos, añadiendo el 0 si es necesario y el valor de los minutos.

El parámetro que se le pasa a la función *"verHora"* es el formato, el cual puede ser FORMATO_LARGO o FORMATO_CORTO. Si se selecciona el largo, se debe añadir el valor de los segundos de la misma forma que las horas y minutos.

Por último, antes de devolver la cadena de caracteres, si el modo es 12h, se añade a la cadena PM si "*AMoPM*" es un 1 lógico o AM si "*AMoPM*" es un 0 lógico.

### *"verDia"*:

La cadena que genera esta función es más simple, ya que solo implica uno de los valores, el día de la semana.

Mediante un switch/case, se añade a la cadena el día que esté almacenado en *"x.diaSem"*.

Como en la anterior, la función recibe el parámetro formato, siendo por defecto el largo. Pero en caso de que se seleccione el formato corto, se eliminan los caracteres de la cadena a partir del tercer carácter. Basta con realizar la operación salida[3]=0.

### *"verMes"*:

Esta función tiene la misma estructura que la anterior, solo que el valor que recibe el switch es el de los meses.

*"verFecha"*:

El modo de añadir los datos a la cadena de la fecha es igual que para el de la hora, con la excepción de que en vez del carácter ":" se utiliza el carácter "/" para separar los datos.

Para esta cadena, el formato corto significa mostrar los años solo desde la década, por lo que se añadiría a la cadena directamente el valor *"x.anno"*, pero en caso de usar el formato largo, se debe sumar 2000 a dicho valor y añadirlo a la cadena sin necesidad de comprobar dígitos.

La peculiaridad de esta función con el resto de funciones "ver", es que recibe un parámetro orden, el cual es un entero que varía entre 1, 2 y 3.

Si el orden es 1, por defecto es este valor, la cadena que devuelve la función será día/mes/año.

Para orden 2, la cadena será año/mes/día. Con objeto de reordenar la cadena se utiliza una función llamada invierte, que cambia los caracteres simétricamente, el primero con el último, segundo con penúltimo, etc. Otra función que se ha creado es la función cambio, la cual realiza un intercambio entre los caracteres de dos posiciones, es igual que la función swap en lenguaje C++.

Finalmente, si el orden es 3, la cadena devuelta será mes/día/año.

"verTemperatura":

Única función "ver" que no devuelve una cadena de caracteres, si no un float, ya que lee los dos registros de temperatura y devuelve su valor.

Esta función no es muy relevante, ya que no tiene relación con los relojes de tiempo real. Se ha incluido en la librería debido a que el dispositivo DS3231 tiene un sensor de temperatura, pero en otros integrados que no tengan sensor, esta función no es válida.

El funcionamiento de esta es similar al de *"leerHora"*, llamando a las mismas funciones de la librería Wire, en este caso comenzando en el registro TEMP_REG1, que es el primer registro de temperatura y solicitando dos bytes, para tomar el valor de los dos registros de temperatura.

La función devuelve el valor del primer registro, parte entera, más el del segundo registro, parte fraccional, multiplicada por 0.25, que es la resolución.

*"alarmasON"*:

Siempre que se quiera activar las interrupciones del RTC o simplemente cambiar el tipo de interrupción, se debe llamar a esta función.

El argumento que recibe es un entero entre 1 y 8, que selecciona el tipo de alarma que se quiere activar.

En primer lugar, la función comprueba que el valor que se le pasa es uno de los tipos, estableciendo el tipo 8 en caso de que no lo sea.

Seguido, se ponen a 1 los bits del registro de control que activan las interrupciones.

Por último, se selecciona el tipo de alarma mediante un switch, los ocho tipos se seleccionan mediante una máscara en el bit más significativo de los siete registros de alarma, y son los siguientes:

1. Alarma cada segundo.
2. Alarma cuando coinciden los segundos.
3. Alarma cuando coinciden minutos y segundos.
4. Alarma cuando coinciden horas, minutos y segundos.
5. Alarma cada minuto.
6. Alarma cuando coinciden hora y minutos.
7. Alarma cuando coinciden día de la semana, hora y minutos.
8. Alarma cuando coinciden día, hora y minutos.

Las máscaras necesarias se presentan en la tabla 3.

| DY/$\overline{DT}$ | ALARM 1 REGISTER MASK BITS (BIT 7) | | | |
|---|---|---|---|---|
| | A1M4 | A1M3 | A1M2 | A1M1 |
| X | 1 | 1 | 1 | 1 |
| X | 1 | 1 | 1 | 0 |
| X | 1 | 1 | 0 | 0 |
| X | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |

| DY/$\overline{DT}$ | ALARM 2 REGISTER MASK BITS (BIT 7) | | |
|---|---|---|---|
| | A2M4 | A2M3 | A2M2 |
| X | 1 | 1 | 1 |
| X | 1 | 1 | 0 |
| X | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 |

Tabla 3. Mascaras de alarmas para el DS3231.

*"escribeAlarma"*:

Para establecer una alarma, se le pasa como argumento a esta función el día, hora, minuto y segundo en que se quiere forzar una interrupción. El orden de los parámetros siempre es este, independiente del tipo de alarma que se quiera establecer. En caso de no necesitar uno de los valores para algún tipo de alarma se puede poner cualquier valor. Y si alguno de los valores no se introduce, por defecto será 0.

La primera sentencia de esta función es llamar a *"am_pm"*, para comprobar el modo del registro hora.

En función del tipo de alarma se escribe en unos registros o en otro. Para los 4 primeros tipos se establece la alarma en los registros de alarma 1, para los otros 4 en los registros de alarma 2.

El procedimiento de escritura en los registros es el mismo para ambos casos. Se almacena en una variable el valor actual del registro en que se va a escribir, obtenido mediante la función *"leeRegistro"*, el sentido de esta lectura es no modificar la máscara que se haya escrito. Tras esto se debe escribir en el mismo registro, este valor leído, con una máscara AND que solo coja el bit más significativo, más el dato codificado.

Para los tipos 1, 2, 3 y 4, se escriben los segundos, los minutos y las horas. En el caso de las horas, si el modo es 24h, se introduce la hora tal cual, pero si el modo es 12h, se debe modificar la hora mediante "*de24a12*" y si "*AMoPM*" es AM se añade a la escritura una máscara OR con el bit 6 a nivel alto, si "*AMoPM*" es PM la máscara OR tendrá tanto el bit 6 como el 5 a nivel alto.

Para los tipos 5, 6, 7 y 8, se escriben los minutos y las horas, como en el caso anterior, y después se añade el día.

*"limpiaAlarma"*:

El uso de esta función es imprescindible para el correcto funcionamiento de las interrupciones, ya que pone a nivel bajo los dos bits menos significativos del registro de estado. Esto dos bits corresponden a los flags de las alarmas, que se ponen a nivel alto cuando se produce una interrupción.

Su ejecución es simple, se lee el registro STATUS_REG y al byte obtenido se le aplica la máscara AND "11111100". El nuevo byte se escribe en el registro.

### *"datoEntre"*:

No hay ninguna función con este nombre en concreto, si no que se refiere a las siete funciones que comprueba sin un dato a escribir en el reloj es correcto.

Estas funciones son booleanas, devuelven true si el valor es válido y false si no lo es.

El parámetro es un entero con el dato que se debe comprobar para cada uno, a excepción de la que comprueba el día, ya que a esta se le debe pasar también el mes, para saber cuál es el día máximo.

Las funciones son *"horaEntre"*, *"hora12Entre"*, *"minsegEntre"*, *"diaEntre"*, *"mesEntre"*, *"annoEntre"* y *"diaSemEntre"*.

### "cambio":

Llamada por la función *"verFecha"* para modificar el orden, esta función recibe tres parámetros, dos son enteros que indican las posiciones a intercambiar, y el tercero es la cadena que se quiere modificar, en este caso será la fecha.

Se crea una cadena auxiliar igual a la recibida, se modifican las posiciones solicitadas y se devuelve la cadena auxiliar.

### "invierte":

Como la anterior, sirve para modificar el orden de la cadena fecha. Esta solo recibe la cadena y mediante dos bucles for, uno en que se van almacenando los valores de las primeras posiciones en una cadena auxiliar y otro en que se almacenan los últimos valores en la misma cadena.

La función devolverá la cadena auxiliar creada.

### *"digitosDato"*:

De esta función se podría prescindir, pero dada la cantidad de veces que se debe realizar la comparación, se ha decidido introducir la comparación en una función. Simplemente se le pasa el dato a comprobar y devuelve true si es menor que 10, false en caso contrario.

***"escribeReloj"*:**

Esta, como la función *"leerHora"*, es imprescindible, ya que es la que se encarga de modificar los registros.

Como argumento recibe la estructura Reloj.

En primer lugar, si la variable AP es false, ya que solamente se pone a true en las funciones *"Modo12h"* y *"escribeHora"*, se almacenarán los valores de actuales en una variable tipo Reloj mediante la función *"leerHora"*. Esto es debido a qué si alguno de los valores a escribir no es válido, se volverá a escribir el actual. Almacenados los valores, se asigna false a AP.

Se prepara la comunicación $I^2C$ con "*Wire.beginTransmission*" y "*Wire.write*", la primera con la dirección del dispositivo y la segunda con 0, para que comience a escribir en el inicio.

A la hora de escribir se realizará dato a dato en el mismo orden que la lectura. Para ello, se llama a la correspondiente función "Entre" para comprobar el dato. En caso de ser válido, se escribe en el registro con "*Wire.write*" el valor del dato, previa codificación.

Como en otras funciones, la variación del registro de la hora dependiendo del modo se debe tener en cuenta, por lo que se comprueba el modo, almacenado en "*AMPM*".

Para el modo 24h se escribe la hora codificada aplicándole una máscara AND que pone a nivel bajo los dos bits más significativos, el bit 7 porque no es relevante y el bit 6 que es el que marca el modo de trabajo.

En el modo 12h se escribirá la hora12 codificada con una máscara AND "00011111", que limpia el bit de modo y de AM/PM, pero se debe comprobar el valor de "*AMoPM*", si es PM( 1 lógico) se aplica una máscara OR que pone los bits 6 y 5 a nivel alto, indicando que es modo 12h y la hora es PM, en caso de que sea AM (0 lógico) la máscara OR solo tendrá a nivel alto el bit 6.

La función no devuelve nada, termina con "*Wire.endTransmission*" para finalizar la comunicación.

**“*decodificar*”:**

Función que recibe un valor entero de un byte, el cual estará en hexadecimal, y devuelve otro entero de un byte con el valor en decimal.

El cálculo que realiza es simple, dividir el valor entre 16 y multiplicarlo por 10, a esto se le debe sumar el resto de la división anterior.

La variable que se devuelve es el valor decodificado mediante el cálculo anterior.

**“*codificar*”:**

Realiza el proceso contrario que decodificar.

Se divide el valor entre 10 y se multiplica por 16, en este caso se suma el resto de la división entre 10.

Devuelve el valor codificado a hexadecimal.

***”leeRegistro”*:**

El argumento que recibe esta función es un byte con la dirección del registro que se quiere leer.

El procedimiento para leer el registro es el mismo que usa *“leerHora”*. Solo que al indicar donde empezar a leer con “*Wire.write”* se debe poner la dirección y se solicita solo un byte mediante “*Wire.requestFrom*”.

El valor leído se guarda en una variable de un byte y es el que se devuelve.

***”escribeRegistro”*:**

Recibe como argumento un byte con la dirección y otro con el valor a escribir.

Se inicia la comunicación y se comienza a escribir el valor en la dirección deseada, para ello se usa “*Wire.write*(dirección)”,

La función no devuelve nada, termina con “*Wire.endTransmission*”.

### *"am_pm"*:

No recibe ni devuelve ninguna variable. Simplemente modifica la variable global "*AMPM*", la cual indica el modo de trabajo.

Se lee el registro de la hora y se le aplica una máscara AND con el bit 6, si el resultado también tiene el bit 6 a nivel alto quiere decir que está activo el modo 12h, por lo que "*AMPM*" pasa a ser ON (1 lógico). En caso contrario "*AMPM*" será OFF (0 lógico).

### "*bit12_24*":

Esta función, como la anterior, tampoco recibe ni devuelve nada.

El objetivo de esta función es asegurar que el bit de modo está a nivel alto si es modo 12h o nivel bajo si es modo 24h.

Se lee el registro de la hora, almacenándose en una variable auxiliar. Si "*AMPM*" es ON, se llama a *"escribeRegistro"* teniendo como argumentos HORA_REG y el valor almacenado en la variable auxiliar con el bit 6 a nivel alto, aplicando una máscara OR "01000000". Y del mismo modo que en *"am_pm"* se comprueba el bit AM/PM.

Si "*AMPM*" es OFF basta con aplicar a la variable auxiliar una máscara AND que pone a nivel bajo los bits 7 y 6.

### *"de12a24"*:

Función necesaria para tener almacenado el valor de la hora en formato 24 hora y así en caso de producirse un cambio de modo, se puede escribir el valor leído sin necesidad de llamar a *"escribeHora"*. Recibe el valor de la hora en modo 12h y lo devuelve en modo 24h.

El procedimiento de la función es simple, si "*AMoPM*" es PM (1 lógico) y la hora es menor de 12, habrá que incrementar a la hora actual en doce horas. En caso de que "*AMoPM*" sea AM (0 lógico), si la hora es 12 pasará a ser 0.

### *"de24a12"*:

Realiza el proceso contrario al anterior, recibe la hora en formato 24h y la devuelve en formato 12h.

En esta ocasión, si la hora es igual o superior a 12, se asigna el valor PM a la variable "*AMoPM*" y en caso de ser superior a 12, la hora se decrementa en doce unidades.

Cuando la hora sea menor que 12, la variable "*AMoPM*" toma el valor AM y no es necesario modificar el valor de la hora.

Si la hora es 0 se modifica su valor, pasando a ser 12.

### *"compruebaCH"*:

El argumento de esta función es la estructura de datos Reloj con variable x, al igual que la variable que devuelve.

Además, esta función acude a tres variables globales, dos enteros, "*horaMas"* y "*horaMenos"*, inicializados a 0, y una booleana "*cH".*

La primera sentencia de la función es declarar una variable "*diaSemana"*, que se empleará para almacenar el valor del día de la semana obtenido por *"calculaDiaSem"*, de esta manera, se puede realizar una comparación para que en caso que el día sea Domingo, definido como el 7 en el código, el día pase a ser 0.

Continuando, se comprueba si "*horaMas"* u "*horaMenos"* son 0, en caso de serlo alguna de las dos, se debe calcular su valor, ya que es el día que se aplicará el cambio de hora.

El cálculo se realiza obteniendo el día de la semana en que cae el día 31, de marzo para "*horaMas"*, de octubre para "*horaMenos"*. Al llamar a *"calculaDiaSem"* se debe tener en cuenta que mes es el actual, porque si el mes de marzo u octubre ha pasado, se calculará "*horaMas"* u "*horaMenos"* respectivamente para el año siguiente.

Las variables "*horaMas"* y "*horaMenos"* serán 31 menos el día de la semana calculado. De ahí que interese que si es Domingo el valor del día de la semana sea 0, ya que los cambios de hora se producen el último domingo de marzo y de octubre.

Tras esto, en la misma condición, se calcula "*cH*", que será true si se ha dado un retraso en la hora y lo siguiente es un adelanto, es decir, de

"*horaMenos*" a "*horaMas*". Si ha habido un adelanto y lo próximo es un retraso, de "*horaMas*" a "*horaMenos*", "*cH*" será false.

Una vez conocidas las variables que indicarán cuando se produce un cambio de hora, se añaden las condiciones para cada caso.

Si "*cH*" es false, el mes es marzo, el día coincide con "*horaMas*" y la hora son las 2, se incrementará la hora y "*cH*" pasa a ser true. Se debe recalcular "*horaMas*" y escribir el nuevo valor de hora en su registro mediante *"escribeReloj"*.

Si no ocurre lo anterior y se dan las siguientes condiciones, "*cH"* es true, el mes es octubre, el día coincide con "*horaMenos"* y la hora es 3, se decrementa la hora y "*cH"* pasa a ser false. Como en el caso anterior, se recalcula "*horaMenos"* y se escribe el nuevo valor en su registro.

Se añade a la estructura Reloj el valor de "*cH"* y la función devuelve la variable x.

### *"calculaDiaSem"*:

La última función de la librería es la encargada de obtener el día de la semana mediante los parámetros recibidos de día, mes y año.

El cálculo se basa en la congruencia de Zeller y son necesarios los siguientes ajustes:

- El año debe estar completo, se le suma 2000.
- Si el mes es enero o febrero, se incrementa en 10 unidades, pasando a ser 11 y 12 respectivamente, y el año se decrementa en una unidad.
- Para el resto de meses se debe decrementar en dos unidades.
- Se obtiene el parámetro "a" como el resto de la división del año entre 100 y el parámetro "b" con el valor entero de la división.

El cálculo del día de la semana utiliza la siguiente formula:

$$700 + \frac{26 \times \text{mes} - 2}{10} + dia + a + \frac{\text{a}}{4} + \frac{\text{b}}{4} - 2 \times b$$

El resultado de dicha fórmula se divide entre 7 y el resto de esta operación es el día de la semana, que se almacena en una variable. En caso de ser domingo, el cálculo dará 0, por lo que se debe cambiar el valor a 7, que es el valor definido en la librería. Finalmente se devuelve el valor del día de la semana.

# 5- Diseño e Implementación de un programa para controlar tarifas eléctricas

Este apartado del trabajo se dedicará al desarrollo de un programa realizado para aplicar las funciones de la librería a un ejemplo práctico.

## 5.1- Objetivo:

El problema que debe solucionar el programa es el de detectar la tarifa eléctrica que corresponde a la franja horaria de la fecha actual.

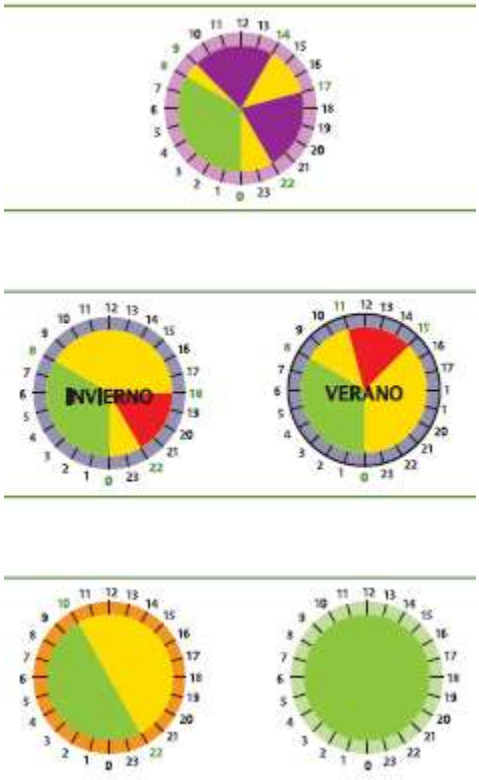
Las tarifas dependen del tipo de día. Existen cuatro tipos de día:

- Días tipo bajo. Son 136 días del año y a ellos se asocia la tarifa más baja, durante todo el día es tipo valle.
- Días tipo medio. 80 días del año coinciden con este tipo. Estos días cuentan con dos tarifas de igual duración, medio día la tarifa valle y el otro medio, la tarifa llano.
- Días tipo alto. 80 días, que se dividen en tres tarifas, valle, llano y punta 100%. La distribución horaria es distinta en verano que en invierno.
- Días tipo pico. 70 días que tienen la tarifa más cara. Se dividen en tres tarifas, valle, llano y punta 300%.

El tipo de día se puede ver en el calendario de la figura 53, la cual es un calendario de 2008, pero que es válido, ya que la distribución de días se hace por festivos, días de la semana, meses y semanas de los meses.


Figura 53.

La distribución de tiempos de las tarifas en cada tipo de día se muestra en la figura 54.




Cada diagrama corresponde a un tipo de día. El de más arriba son los días de pico, los dos del medio son los días altos, que se dividen en invierno y verano. En la parte baja de la imagen están los días de tipo medio a la izquierda y los días tipo bajo a la derecha.

El color verde indica la tarifa valle, el amarillo corresponde a la tarifa llano y el rojo y morado corresponden a la tarifa punta, siendo el rojo la tarida punta 100% y el morado la tarifa punta 300%.

Figura 54.

## 5.2- Desarrollo del software:

Como introducción, el funcionamiento del programa consistirá en ir estableciendo alarmas en las horas que se produce un cambio de tarifa. Para ello, debe conocerse el tipo de día, el cual se comprobará cada día, cuando se alcance el estado 0, que lleva asociada la alarma a las 00:00:00.

Con el proceso anterior se almacena que tipo de tarifa corresponde al instante actual. Con la alarma 2 se establecerán las horas de inicio y fin de suministro.

### Variables, funciones y parámetros necesarios en el programa:

Se definen los tipos de día y las tarifas como los enteros 1, 2, 3 y 4 en el siguiente orden BAJO, MEDIO, ALTO y PICO, para los días, VALLE, LLANO, PUNTA100 y PUNTA300, para las tarifas.

También se define INVIERNO como false y VERANO como true.

La lista de las variables es la siguiente:

- Variable binaria “época”, almacenará si actualmente es horario de verano o de invierno.
- Variable tipo entero “estado”, la cual almacenará por dónde va la secuencia, para realizar el seguimiento. Cada cambio de tarifa se producirá en un estado.
- Es necesario un vector de enteros “festivos” que contendrá las fechas festivas, sucediéndose día y mes para los días festivos oficiales.
- Un entero para identificar el tipo de día “tarifadia”, se inicializa a BAJO.
- Como la anterior, para identificar la tarifa se usa un entero “tarifa”, inicializado a VALLE.
- Se utiliza una variable tipo Reloj “y”, que pertenece a la librería, para realizar las comprobaciones de fecha actual.
- La variable booleana “suministro”, indicará si este está activo o no.
- Mediante los enteros “h_ini”, “m_ini”, “h_fin” y “m_fin”, se gestionan los tiempos de inicio y final del suministro. Estos valores se establecen en el fichero de texto.
- Para indicar que se ha activado una alarma y por tanto se debe cambiar de estado, se declara la variable “cambio”, la cual es binaria y debe ser volatile, ya que se va a variar dentro de la interrupción. Esto fuerza a que cada vez que se use la variable “cambio”, se deba comprobar si ha variado, dado que si se está ejecutando la función principal, el valor de esta variable es el que tiene al inicio de la ejecución, y al producirse una interrupción, el programa se ejecuta desde el punto en que se paró, instante en el que la variable tenía un valor al que retornaría en caso de haberse cambiado en la llamada de la interrupción.
- Una variable binaria y volátil llamada “manual”, indicará si el suministro se activa mediante las alarmas de inicio y fin establecidas, o de manera manual.

### 5.2.1- Diagramas de flujo:

Para realizar los cambios de estado, se esperará a una interrupción, la cual es forzada por las alarmas del RTC, dicha interrupción llama a la función “interrup()”. Esta función, simplemente pondrá el valor de la variable volátil “cambio” a true.


Figura 55.

Función “tarifadia()” para comprobar que tipo de día es. Constará de varias sentencias condicionales. No necesita ningún argumento y el valor que devuelve es un entero, que será 1, 2, 3 o 4. En la figura 56 se muestra el diagrama de flujo. No se muestran las sentencias condicionales, que utilizan una gran cantidad de operadores OR y AND, para la comparar la fecha actual, con las fechas de cada tipo de día.

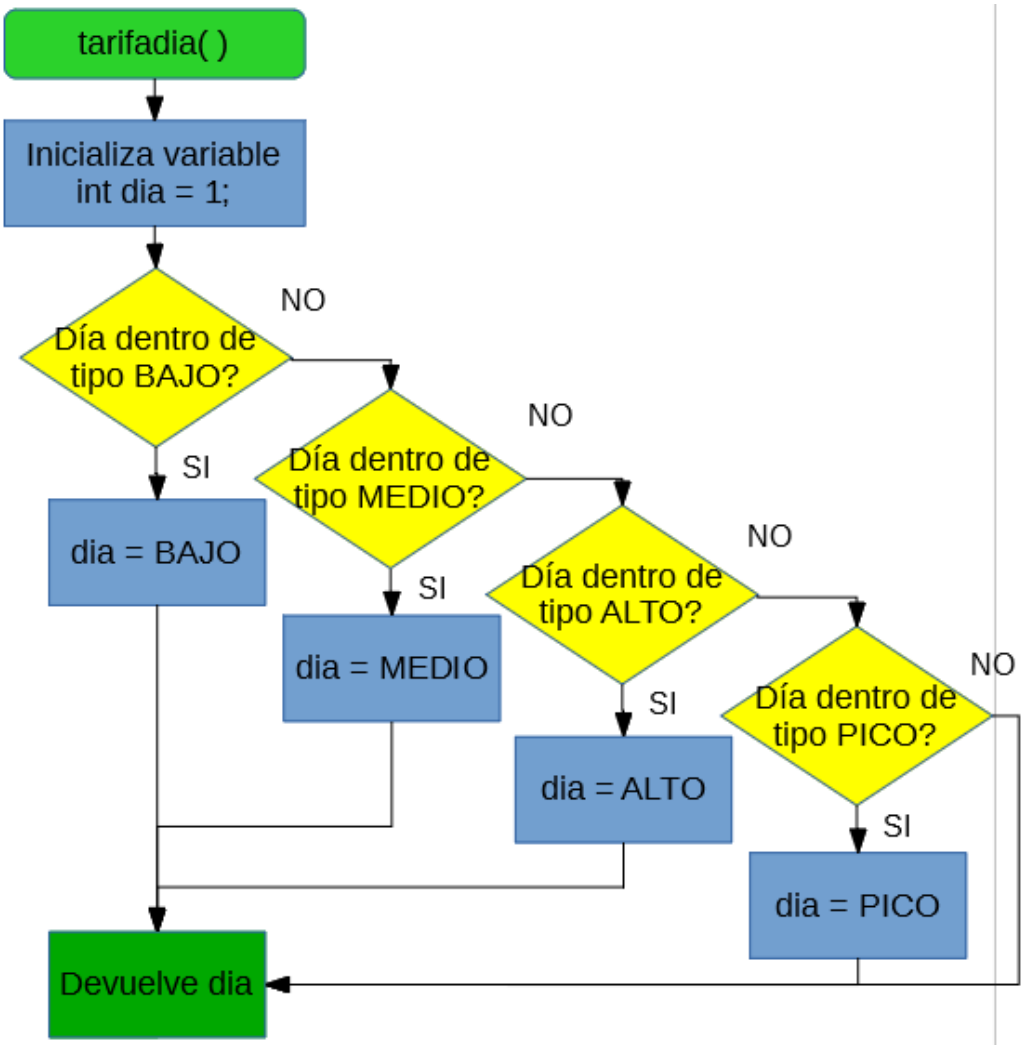



Figura 56.

Es necesaria una función que lea los datos de un fichero de texto. Estos datos corresponden a las fechas que modifican alguna de las tarifas y que pueden variar, en concreto los días festivos, que en ocasiones se mueven en el calendario, por ejemplo, para pasarlos de un domingo a un lunes. Tras los días festivos se incluye una hora que será la de inicio de suministro y otra que es la de fin del mismo, que establecerán la alarma 2 del RTC.

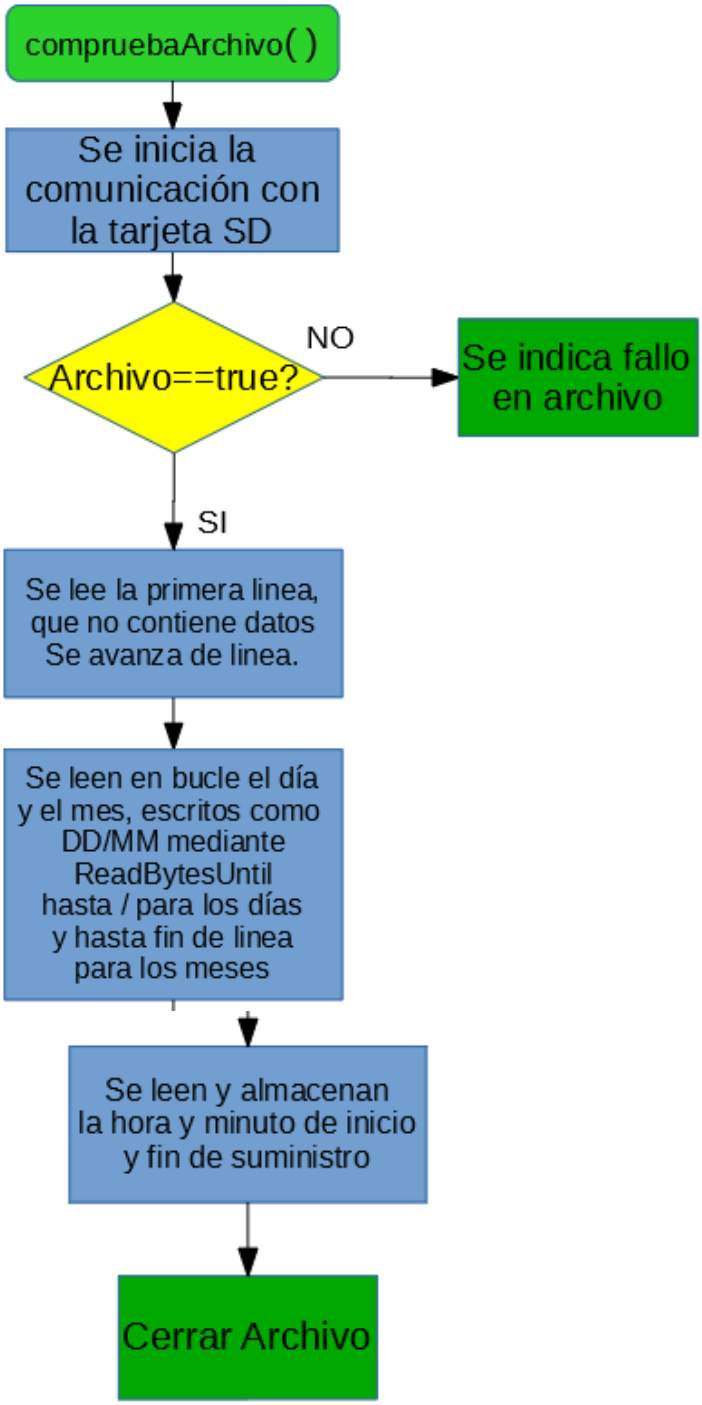



Figura 59.

Como cualquier programa de Arduino, tiene una función “setup”, donde se declaran las sentencias iniciales. Su diagrama de flujo se presenta en la Figura 57. Las sentencias que se sucederán son:

- Inicializar y limpiar el display LCD.
- Almacenar el valor actual de la fecha en la variable y.
- Establecer el modo horario, se hará en modo 24h.
- Limpiar los *flags* de las alarmas.
- Establecemos la alarma 1 de tipo 4 que variará en cada estado. Se utiliza esta alarma porque la interrupción es independiente del día, depende de la hora, minuto y segundo. Se fija a las 00:00:00, para que la sucesión de estados comience en un nuevo día.
- Se habilita la interrupción 0 cuando se produzca un flanco descendente.

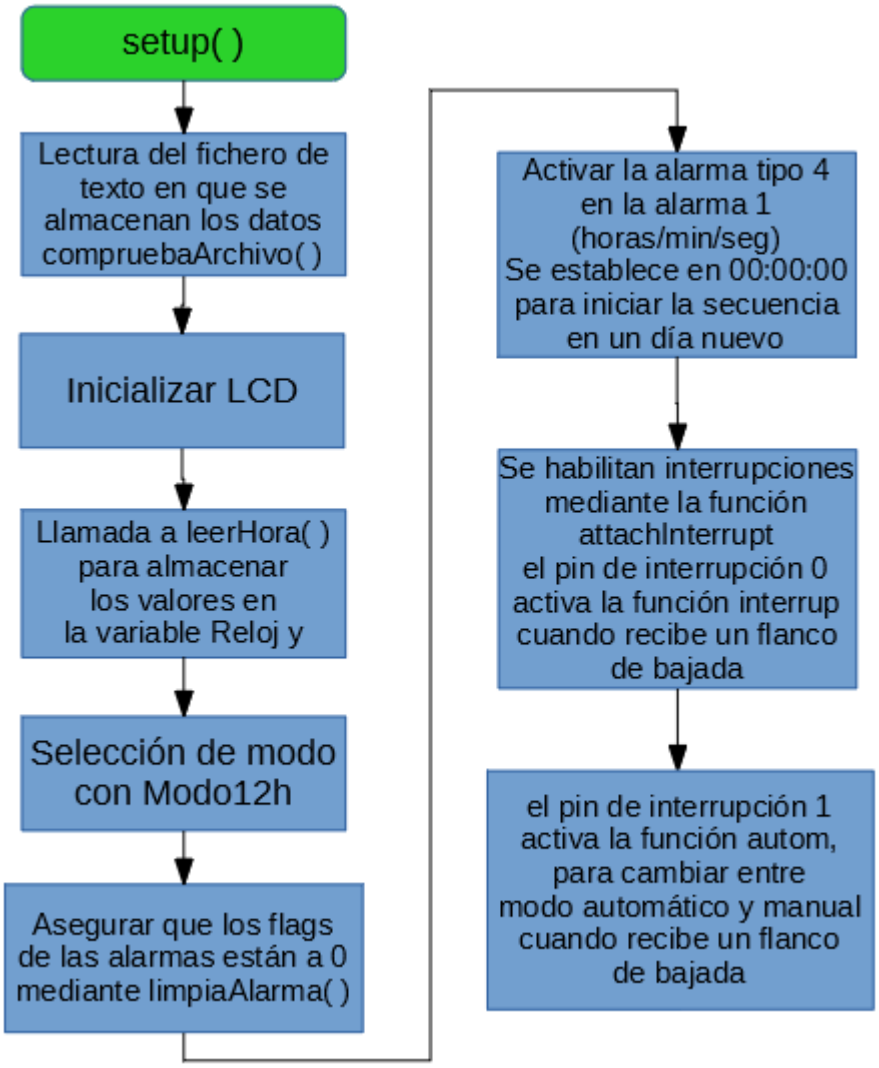



Figura 57.

Función principal, que se ejecutará en bucle "loop". Tiene dos sentencias condicionales. Una primera que comprueba si está en el estado 1, estado en que se comprueba el tipo de día. La segunda se ejecuta si "cambio" es true, es decir, si se ha producido una interrupción que ha modificado a "cambio".

El diagrama de flujo se puede ver en la Figura 58, en él se puede entender el funcionamiento, pero de todas formas a continuación se redacta el proceso.

Como se ha mencionado ya, la primera sentencia condicional se ejecuta si "estado" es 1, en caso de que esto ocurra, se asignará a "tarifa" el valor VALLE. Será necesario almacenar los valores de fecha en la variable Reloj "y", llamando a la función "*leerHora*()". Se hace una llamada a la función "tarifadia()" y el valor devuelto se le asigna a la variable "tarifadia", la cual se le pasa a una función switch, que dependiendo del tipo de día asigna un valor a estado y establece la siguiente alarma.

- Tipo BAJO: estado 0, alarma programada a las 0:00:00.
- Tipo MEDIO: estado 2, alarma programada a las 10:00:00.
- Tipo ALTO: estado 4, alarma programada a las 8:00:00.
- Tipo PICO: estado 7, alarma programada a las 8:00:00.

Tras esta sentencia, se ejecute o no su proceso, se comprueba la segunda condición. Si la variable "cambio" es verdadera significa que se ha producido una interrupción debida a alguna de las alarmas definidas en un estado anterior y se ejecuta una sentencia switch con el valor de estado. Esto comprueba en que estado se encuentra actualmente el proceso.

- Estado 0: próximo estado 1, tarifa no varía, sigue siendo VALLE.
- Estado 2: próximo estado 3, tarifa cambia a LLANO. Alarma programada para las 22:00:00.
- Estado 3: próximo estado 0, tarifa cambia a VALLE. Alarma programada a las 0:00:00.
- Estado 4: próximo estado 5, tarifa cambia a LLANO. Alarma programada a las 11:00:00 si es verano (variable "epoca" es true) o a las 18:00:00 si es invierno (variable "epoca" es false).
- Estado 5: próximo estado 6, tarifa pasa a PUNTA 100%. Alarma programada a las 15:00:00 si época=VERANO o a las 22:00:00 si no.
- Estado 6: próximo estado 0, tarifa cambia a LLANO. Alarma a 0:00:00.
- Estado 7: próximo estado 8, tarifa pasa a LLANO. Alarma a las 9:00:00.
- Estado 8: próximo estado 9, tarifa PUNTA 300%. Alarma a las 14:00:00.
- Estado 9: próximo estado 10, tarifa a LLANO. Alarma a las 17:00:00.
- Estado 10: próximo estado 6, tarifa PUNTA 300%. Alarma a las 22:00:00

A cada tipo de día le corresponde una serie de estados, el estado 0 a los días tipo BAJO, el 2 y 3 a los días tipo MEDIO, el 4,5 y 6 al tipo ALTO y del 7 al 10, añadiendo el estado 6 al final, corresponden a los días de PICO.

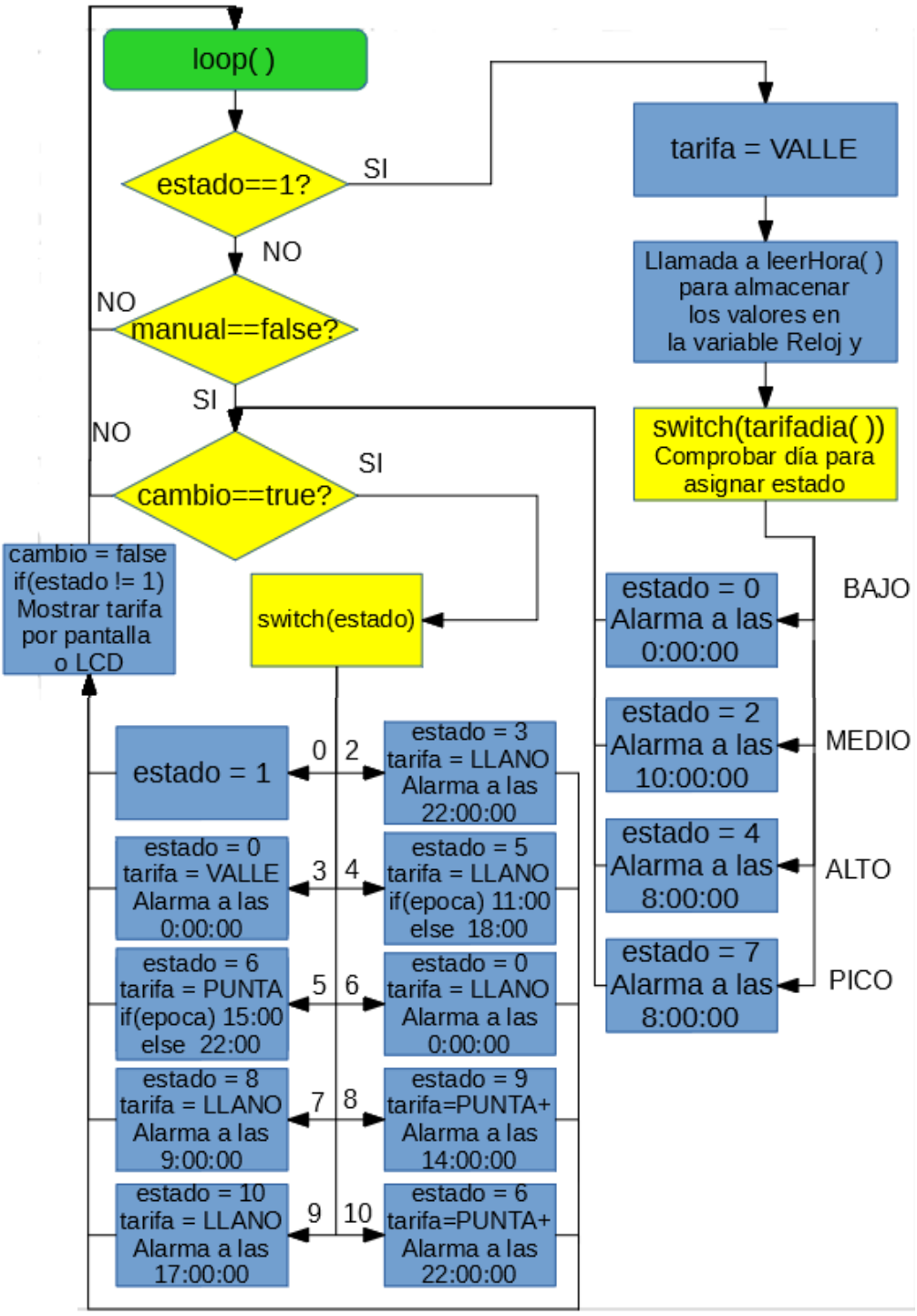



Figura 58

### 5.2.2- Diagrama de estados:

En la siguiente figura se muestra el diagrama de estados que representa el proceso anteriormente explicado.

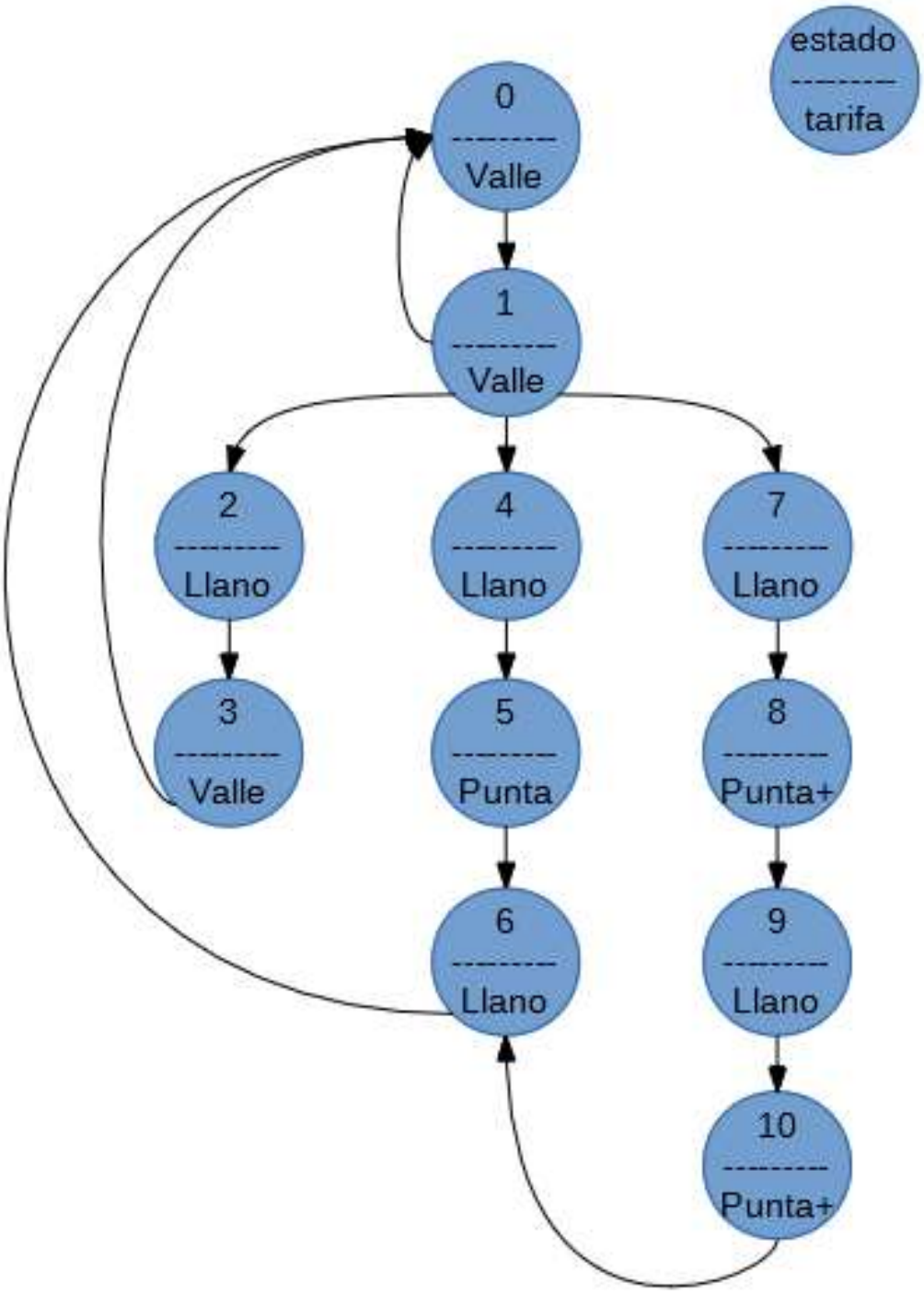



Figura 60.

## 5.3- Desarrollo del código:

Se incluyen las siguientes librerías:

- Wire.h, necesaria para la comunicación I²C del RTC y del display LCD.
- LCD.h, añade las funciones necesarias para el manejo del display.
- RTC3231.h, es la librería creada en este proyecto, que nos permitirá controlar el reloj de tiempo real.
- LiquidCrystal_I2C.h, esta librería es necesaria incluirla, ya que el display que se va a utilizar, lleva conectado una placa para la comunicación mediante el BUS I²C.

Aparte del resto de definiciones comentadas en el punto anterior, se debe declarar la constante LCD_DIR 0x27, la cual indica la dirección del display.

La primera sentencia declara la variable para controlar el display mediante I²C, indicando los pines de datos del mismo. La línea de código es la siguiente:

```
LiquidCrystal_I2C lcd(LCD_DIR, 2, 1, 0, 4, 5, 6, 7, 3, POSITIVE);
```

Tras esta inicialización, declaradas todas las variables, viene la función "setup". En ella, como se ha mencionado en el diseño, se inicializa el display, con los caracteres por línea y las líneas que tiene, y se limpia, para ello se emplea:

`lcd.begin (16,2);` Indica que tiene 2 líneas con 16 espacios.

`lcd.setBacklight (ON);` Sirve para encender la luz del display.

`lcd.clear ();` Limpia el contenido actual del display.

Se deben habilitar las interrupciones, para que al detectar un flanco de bajada en el pin de interrupciones 0 o 1, se llame a la función "interrup" o "autom":

```
attachInterrupt (0, interrup, FALLING);
attachInterrupt (1, autom, FALLING);
```

El resto de sentencias de esta función "setup", son llamadas a las funciones desarrolladas en la librería de este trabajo:

compruebaArchivo(); Esta función lee un fichero y almacena diversos datos.

`y = rtc.leerHora();` Lectura y almacenamiento de fecha.

`rtc."Modo12h"(OFF);` Selección del modo horario.

`rtc."limpiaAlarma"();` Poner a 0 los flags de las alarmas.

*rtc.alarmasON(4);* Activar alarmas y seleccionar tipo 4 (hora/min/seg).

*rtc.escribeAlarma(0,0,0,0);* Primera alarma al iniciarse un día (estado=0).

*rtc.alarmasON(6);* Activa la alarma tipo 6 (hora/min).

*rtc.escribeAlarma(0,h_ini,m_ini,0);* Alarma para el inicio y fin del suministro.

La función que consulta el tipo de día, asignará un valor a la variable entera "dia" y lo devolverá. Las condiciones de cada día, son las indicadas en la tabla de la figura 53. A continuación se muestran las sentencias condicionales:

*if (y.diaSem==DOMINGO || (y.mes==4 && y.dia>27) || (y.mes==5 && y.dia<25) || (y.mes==8 && y.dia>3 && y.dia<23) || (y.diaSem==SABADO && y.mes>2 && y.mes<12) || recorre_festivos(y.dia,y.mes))*

Dada una de estas seis condiciones, el día será tipo bajo.

*else if ((y.mes==3 && y.dia==31) || (y.mes==4) || (y.mes==5) || (y.mes==6 && y.dia<16) || (y.mes==8 && y.dia>24) || (y.mes==9 && y.dia>14) || (y.mes==10 && y.dia<25) || (y.mes==12 && y.dia==8))*

En caso de que no se cumpla ninguna de las primeras condiciones, se comprueban ocho nuevas, que en caso de cumplirse alguna determina el día de tipo medio.

*else if (y.mes==3 || y.mes==6 || y.mes==7 || y.mes==8 || y.mes==9 || y.mes==10 || (y.mes==11 && y.dia<16) || (y.mes==11 && (y.dia==5||y.dia==26)) )*

Si tampoco se cumple alguna de las anteriores condiciones, se comprueban estas ocho, que incluyen el resto de días de los meses ya comprobados y algún día suelto. Si se cumple una de las condiciones, el día será tipo alto.

Por último, si no se ha cumplido ninguna de los condicionales, el tipo de día corresponderá al tipo pico.

Esta función devolverá la variable en que se almacena el tipo de día. Siendo uno de los valores indicados en el diseño del punto 5.2.1.

La otra función necesaria para el correcto funcionamiento del código principal, es la función "interrup".

Esta función no devuelve nada, simplemente asignará a la variable "cambio" el valor true. Al estar esta variable binaria a nivel alto, se ejecutará el cambio de estado en la función principal "loop".

La función que leerá los datos del fichero de texto y los almacenará en variables, es la función compruebaArchivo. Esta función no recibe ningún argumento ni devuelve ningún valor. Simplemente conecta con el fichero de texto, llamado "Archivo.txt", y siguiendo un orden recorrerá el fichero completo, almacenando cada valor en un buffer, que posteriormente se debe convertir en un entero y almacenar en un vector llamado "festivos". A continuación, se explican varias líneas del código:

```
char linea[10];
```
Variable utilizada para gestionar que leer de una línea.

```
char buffer[3];
```
En este buffer se almacena la lectura actual del dato.

*Se establece como salida el pin correspondiente a SS, de selección de esclavo.*

```
pinMode(10, OUTPUT);
```

Comprobación de que se ha abierto correctamente el fichero de texto.

```
if (!SD.begin(10)) {
    Serial.println("Se ha producido un fallo al iniciar la comunicación");
    return;
}
```

```
Archivo = SD.open("datos.txt");
```
Sentencia que abre el fichero.

*Si el archivo se ha abierto correctamente se almacenan los datos.*

```
if (Archivo) {
```

Mediante la función readBytesUntil se lee hasta el salto de línea y con la función seek y la función position, se pasa a la siguiente línea, esto es debido a que la primera línea no contiene datos, si no la indicación del formato.

```
Archivo.readBytesUntil('\r', linea, sizeof(linea));
Archivo.seek(Archivo.position() + 1);
```

Para leer y almacenar cada fecha festiva, se debe recorrer un bucle de la mitad del tamaño del vector festivos, ya que el tamaño de este es un máximo de 24 enteros, es decir, de 12 datos de día y 12 de mes.

```
for(int i=0;i<sizeof(festivos)/2;++i)
{
  buffer[0,1]='\0';
```
Igualando al carácter \0, se vacía el buffer.

Con las siguientes dos líneas se lee el primer dato hasta el carácter '/', por lo que el carácter leído es el día, el cual se almacena en el buffer. Después se transforma a un entero el valor del buffer y se almacena en la posición i del vector festivos.

```
Archivo.readBytesUntil('/', buffer, sizeof(buffer));
festivos[i]=atol(buffer);          La función atol pasa de carácter a entero.
buffer[0,1]='\0';                  Volvemos a limpiar el buffer.
```

Las próximas dos líneas realizan el mismo proceso, pero leyendo desde el carácter '/' hasta el fin de línea, para almacenar en este caso el mes.

```
Archivo.readBytesUntil('\r', buffer, sizeof(buffer));
festivos[i+1]=atol(buffer);
Archivo.seek(Archivo.position() + 1);          Sentencia que pasa de línea.
}
```

Al finalizar el bucle se vuelve a limpiar el bucle.

```
buffer[0,1]='\0';
```

Ahora, como en el caso anterior, se salta la primera línea. En esta ocasión no es necesario un bucle, ya que solo hay dos líneas, una con la hora de inicio y otra con la hora de fin, ambas con el formato hh:mm.

```
Archivo.readBytesUntil('\r', linea, sizeof(linea));
Archivo.seek(Archivo.position() + 1);
Archivo.readBytesUntil(':', buffer, sizeof(buffer));
h_ini=atol(buffer);          buffer[0,1]='\0';
Archivo.readBytesUntil('\r', buffer, sizeof(buffer));
m_ini=atol(buffer);
Archivo.readBytesUntil('\r', linea, sizeof(linea));
Archivo.seek(Archivo.position() + 1);
buffer[0,1]='\0';
Archivo.readBytesUntil(':', buffer, sizeof(buffer));
h_fin=atol(buffer);          buffer[0,1]='\0';
```

```
    Archivo.readBytesUntil('\r', buffer, sizeof(buffer));

    m_fin=atol(buffer);

  Archivo.close();  }
```

En la función “loop” que se ejecuta en cada ciclo de Arduino, se comprueba si el proceso está en el estado 1, ya que en este estado es cuando se realiza la llamada a la función “tarifadia”, comprobando el tipo de día y asignando el siguiente estado y la próxima interrupción en función de este.

En caso de ser el estado 1, se le asigna a tarifa el valor VALLE, se leen y almacenan los valores de la fecha y se comprueba el tipo de tarifa. El código es el siguiente:

```
if(estado==1){

y = rtc.leerHora();

tarifa = VALLE;

tarifadia = tarifaDia();

switch (tarifadia)

  {
    case BAJO:
      estado = 0;
      rtc.escribeAlarma(0,0,0,0);
    break;

    case MEDIO:
      estado = 2;
      rtc.escribeAlarma(0,10,0,0);
    break;

    case ALTO:
      estado = 4;
      rtc.escribeAlarma(0,8,0,0);
    break;

    case PICO:
      estado = 7;
      rtc.escribeAlarma(0,8,0,0);
    break;
  }
```

Para realizar la comprobación, se muestra en una línea del display la fecha y en otra el tipo de día y de tarifa. Después se añade al fichero de texto Registro.

Las líneas que ejecutan estos procesos son las que siguen:

```
lcd.clear();
lcd.setCursor(0,0);
lcd.print("Es dia "); lcd.print(y.dia); lcd.print(", mes ");lcd.print(y.mes);
lcd.setCursor(0,1);
lcd.print("Tipo:"); lcd.print(tarifadia); lcd.print(" Tarifa:");lcd.print(tarifa);
}
```

Aquí termina el proceso relacionado al estado 1.

La segunda comprobación que se hace en bucle es ver si la variable “cambio” es *true*. En caso de serlo, como se indicó en el diseño del punto 5.2.1, se debe comprobar el estado actual, para establecer el próximo estado y la siguiente alarma, pero antes se ejecutarán las siguientes sentencias:

```
y = rtc.leerHora();      Se almacena la fecha, para la comprobación de época.
rtc.limpiaAlarma();      Limpia el flag de la alarma que ha producido el cambio.
```

```
if(y.diaSem==DOMINGO)
{
    tarifa = VALLE;
     if(y.dia>24 && (y.mes==3||y.mes==10))
     {
       if(y.mes==3)  epoca=VERANO;
       else epoca=INVIERNO;
     }
}
```

Esta comprobación se realiza para almacenar el valor de la variable “época”, la cual es necesaria para determinar los cambios de estado en caso de que el tipo de día sea pico. Simplemente, en caso de ser el último domingo de marzo se produce el cambio a horario de verano y si es el último domingo de octubre se cambia a invierno.

La condición se realiza con domingo, ya que todos los domingos son día tipo bajo, por lo que no hay que variar el estado ni la alarma. Si no se produce esta condición, se procede a la comprobación del estado actual anteriormente mencionada, cuyo código es el siguiente:

```
switch(estado) {
  case 0:
   estado = 1;
   break;
  case 2:
   tarifa = LLANO;
   rtc.escribeAlarma(0,22,0,0);
   estado = 3;
   break;

  case 3:
   tarifa = VALLE;
   rtc.escribeAlarma(0,00,0,0);
   estado = 0;
   break;

  case 4:
   tarifa = LLANO;
   if(epoca)
     rtc.escribeAlarma(0,11,0,0);
   else
     rtc.escribeAlarma(0,18,0,0);
   estado = 5;
   break;

  case 5:
   tarifa = PUNTA100;
   if(epoca)
     rtc.escribeAlarma(0,15,0,0);
   else
     rtc.escribeAlarma(0,22,0,0);
   estado = 6;
   break;

  case 6:
   tarifa = LLANO;
   rtc.escribeAlarma(0,00,0,0);
   estado = 0;
   break;
```

```
    case 7:
      tarifa = LLANO;
      rtc.escribeAlarma(0,9,0,0);
      estado = 8;
    break;

    case 8:
      tarifa = PUNTA300;
      rtc.escribeAlarma(0,14,0,0);
      estado = 9;
     break;

    case 9:
      tarifa = LLANO;
      rtc.escribeAlarma(0,17,0,0);
      estado = 10;
     break;

    case 10:
      tarifa = PUNTA300;
      rtc.escribeAlarma(0,22,0,0);
      estado = 6;
     break;
  }
}
```

Como se puede observar en estas líneas de código, la comprobación de estado conlleva a un cambio de estado y, por lo tanto, a un cambio de tarifa y de alarma. Tras esto, se debe reestablecer de nuevo la variable “cambio” a *false*, para que entre en esta sección del código tras la siguiente interrupción.

Para la prueba de funcionamiento se muestra el tipo de día y la tarifa en la segunda línea del display. En la primera se sigue mostrando la fecha que se establece en el estado 1.

## 5.4- Diseño hardware:

En este apartado se explican los componentes utilizados para realizar las pruebas del programa de ejemplo.

Los componentes empleados son los siguientes:

- Placa Arduino UNO. Para nuestro programa hay que tener en cuenta que los pines de interrupción son el pin 2 para la interrupción 0 y el pin 3 para la interrupción 1.
- Un módulo ZS-042, el cual tiene el integrado DS3231, el RTC para el que se ha diseñado el programa. Además, este módulo viene preparado para incluir la batería de alimentación alternativa para el reloj. A la hora de conectar una batería a este módulo, hay que prestar atención a que sea la correcta, ya que está diseñado para que se le conecte una batería recargable. Esta batería está unida a una resistencia y un diodo, para que en caso de conectarse a la alimentación primaria, la batería se cargue. En nuestro modelo de prueba se ha retirado dicha resistencia, para poder emplear una pila no recargable.

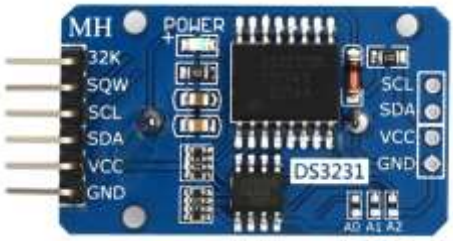



Figura 61. Módulo ZS-042.

- Display LCD con YwRobot. Este display se controla mediante el bus $I^2C$, por lo que solo necesita dos pines, el de transferencia de datos y el de reloj, más los pines de alimentación y tierra.


Figura 62. Display LCD con YwRobot.

- Lector de tarjetas micro SD. Este módulo se conecta mediante la comunicación serie bidireccional SPI, dicho protocolo requiere un pin MOSI, un pin MISO, una señal de reloj CLK y un pin SS. Resumido brevemente, este protocolo opera con un maestro y uno o varios esclavos. Mediante la señal SS se selecciona el esclavo con el que se habilita la comunicación y mediante la señal CLK se comienzan a enviar simultáneamente las señales MOSI (del maestro al esclavo) y MISO (del esclavo al maestro). Con cada pulso de reloj se transfiere un bit. Para el caso de Arduino UNO los pines están definidos como se muestra a continuación:
    - MOSI → pin 11.
    - MISO → pin 12.
    - CLK → pin 13.
    - SS → pin 10.

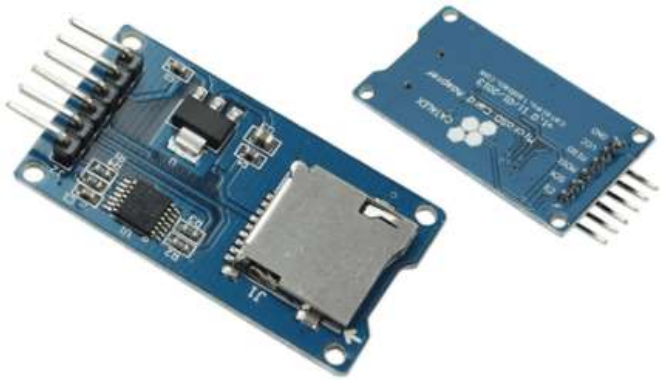


Figura 63. Módulo lector de micro SD.

El montaje se muestra en la Figura 64.

Las conexiones rojas son las de alimentación, de los 5V de Arduino va a los pines VCC del RTC y del display.

La unión entre los terminales de tierra de la placa Arduino, el display y el RTC es la negra.

Las líneas naranjas van de los pines SCL del display y el RTC al pin A5 de Arduino, el cual es el pin de reloj para la comunicación I²C.

En azul, las líneas de datos del bus I²C, del pin A4 de Arduino, el cual se emplea para la transmisión de datos serie, a los pines SDA del display y el RTC.

Por último, en verde, la conexión del pin SQW del RTC al pin 2 de Arduino, pin establecido para la interrupción 0. La salida SQW del RTC estará a nivel alto hasta que se active alguno de los flags de alarma.

En este esquema falta el lector de tarjetas, que como se ha indicado se conecta en los pines del 10 al 13.

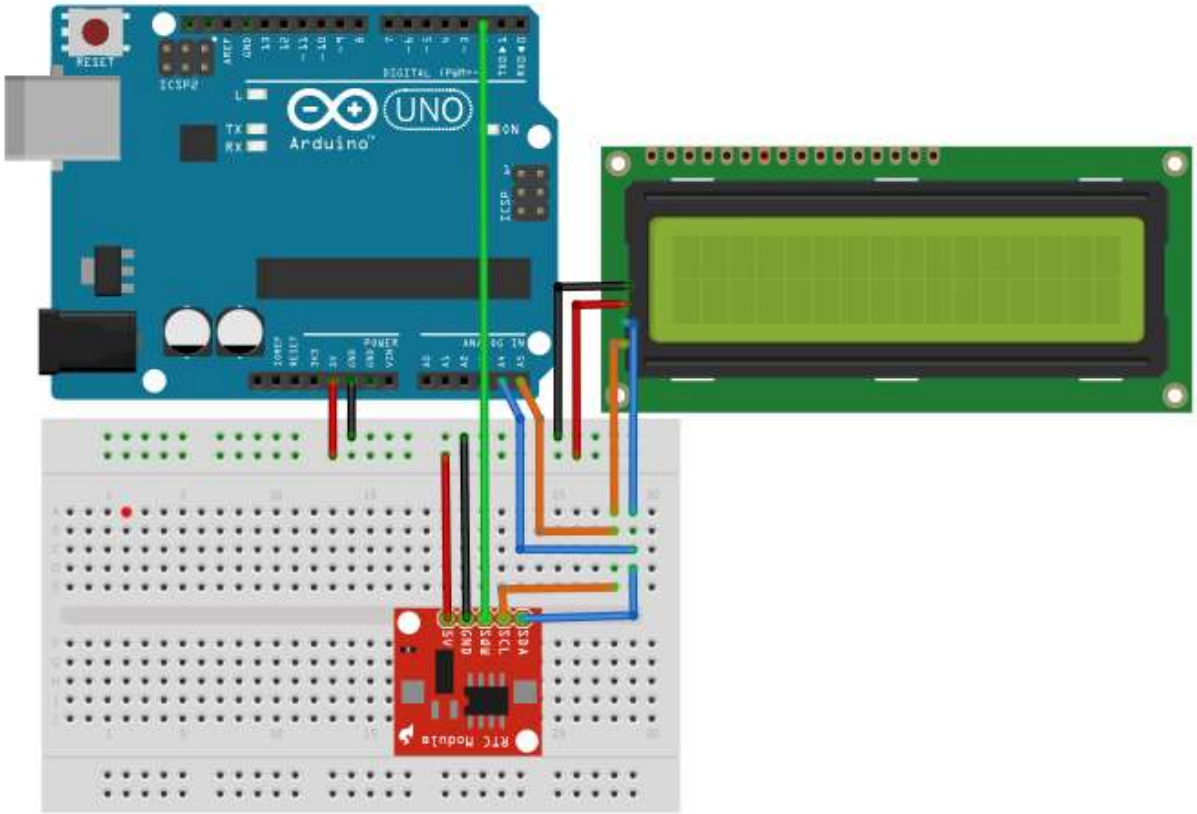



Figura 64. Montaje de Arduino.

# 6- Conclusiones y líneas futuras

En resumen, el proyecto se ha dedicado al desarrollo de las funciones necesarias para el control de un RTC. A continuación, se realiza una lista de las funciones imprescindibles para cualquier RTC:

- Lectura de los registros.
- Escritura de registros.
- Modificar fecha y hora.
- Activación de alarmas, así como selección de tipo de alarma.
- Establecimiento de la alarma activada.
- Selección y comprobación del modo horario.

En general, estas son las funciones que podrían hacer funcionar el RTC con sus alarmas. Pero para reducir las funciones principales y facilitar la comprensión del código, se han desarrollado funciones secundarias explicadas en esta memoria.

Como conclusión, se comentan las ventajas de la utilización de un reloj de tiempo real, frente a otros métodos alternativos con los que se puede realizar un control temporal. Algunos métodos posibles son:

- Actualizar la hora mediante un servidor en red. Para realizar esta sincronización se ha creado la norma NTP, sin entrar en detalles, este protocolo solicita la hora a un servicio central de Internet. Este método es muy preciso, pero requiere material de mayor valor y un buen acceso a Internet.

- Un control menos preciso se puede realizar mediante software, usando la función millis() o micros() de Arduino. Estas funciones miden el tiempo desde que se ejecutó el primer ciclo del programa. Con distintas funciones, conocida la fecha actual, se podrían ir almacenando futuras fechas.

Al utilizar un dispositivo RTC, aseguramos una medida del tiempo exacta sin necesidad de tener acceso a red, pudiendo controlar el tiempo en lugares donde no hay conexión a Internet.

Con respecto al método software, la utilización de un RTC nos asegura el mantenimiento de la fecha y hora actual en caso de que la alimentación principal se desconecte, ya que posee una batería para la alimentación alternativa. Al usar la función millis(), se corre el riesgo de que Arduino deje de recibir alimentación en un momento dado, reiniciándose por el primer ciclo, lo que haría que el tiempo vuelva a comenzar desde cero.

Una de las principales ventajas por las que se utilizan este tipo de dispositivos, es por las alarmas. Gracias a las interrupciones, no es necesario consultar la hora o fecha en cada ciclo de Arduino, cuando se va a producir un evento, basta con establecer una alarma en esa fecha u hora, y la interrupción provoca la modificación oportuna en el programa.

El empleo de interrupciones libera al microcontrolador de mucha carga de trabajo. En nuestro programa de ejemplo, se puede comprobar, en cada ciclo del programa no se realiza ninguna acción, a no ser que se haya producido una interrupción, entonces ya se realiza la comprobación del estado o en su caso la de la fecha actual, para asignar el tipo de día.

### Líneas futuras:

En este apartado se abordan ligeramente posibles mejoras en la librería, así como ideas de proyectos en los que se podría utilizar la librería desarrollada.

Un posible añadido a la librería, sería el control de una pequeña base de datos, en la que se puedan almacenar distintos valores de fecha y hora, siendo útil para la gestión de alarmas, ya que los dispositivos RTC tienen un límite de alarmas, en el caso del dispositivo DS3231 empleado en este proyecto, dos alarmas.

Esta base de datos ordenaría las horas y fechas cronológicamente, de esta manera, al producirse una alarma se acudiría a la base de datos para establecer la más próxima y eliminándola de la base de datos una vez establecida.

La base de datos sería un simple fichero de texto, el cual se podría almacenar en la EEPROM de Arduino, aunque el tamaño que puede almacenar es reducido, o en una tarjeta SD a la cual tendría acceso Arduino mediante el módulo lector de tarjetas utilizado en este mismo trabajo.

Este sistema podría ser útil para procesos que necesiten gran cantidad de interrupciones que se sucederían sin necesidad de solicitar al usuario ser establecida o sin estar incluidas en el código fuente principal.

A continuación, se mencionan brevemente posibles proyectos a interesantes a realizar con un dispositivo RTC y la librería implementada en el presente proyecto:

- Sistema con el que controlar fechas de caducidad de productos. Para realizar este sistema sería necesaria una base de datos, como se ha explicado anteriormente. En esta base de datos se irían añadiendo productos con un identificador y su fecha de caducidad, por ejemplo, mediante una lectura de código de barras. Sería interesante que los datos se ordenen en una pila por orden cronológico, de esta manera, cuando se produzca una alarma y se consulte cual es la siguiente, no debe comprobar toda la lista, si no que valdrá con tomar el primer dato.

- Otra posible aplicación podría ser un sistema de control de riego. De la misma forma que se ha hecho un programa para el control de las tarifas eléctricas, puede realizarse uno para el control de las tarifas de riego. Con esto podrían añadirse condiciones como, regar solo si la tarifa actual es valle, y aprovechando que hay dos alarmas, podría añadirse una alarma con una interrupción que active el riego y establezca una próxima alarma con la misma interrupción, cuando se quiera desactivar el riego. Para ello sería necesaria una variable binaria que indique si el riego está o no activo.

- Como estas, se puede realizar gran cantidad de aplicaciones en las que sea necesario realizar un control de tiempo y de eventos.

# 7- Bibliografía


Arduino.
<https://www.arduino.cc>

Aprendiendo Arduino. (2015). “Qué es Arduino y el Hardware Libre”. Consultado en junio de 2017.
<https://aprendiendoarduino.wordpress.com/2015/03/22/que-es-el-hardware-libre>

Arduino. “Estructura de un programa”. Consultado en junio de 2017.
<http://playground.arduino.cc/ArduinoNotebookTraduccion/Structure>

Aprendiendo Arduino. (2016). “Librerías (Gestor de Librerías)”. Consultado en junio de 2017.
<https://aprendiendoarduino.wordpress.com/2016/03/31/librerias-gestor-de-librerias>

Meneu, J. J. (2015). “Los relojes en tiempo real (RTC) nunca duermen”. Consultado en mayo de 2017.
<https://www.arrow.com/es-mx/research-and-events/articles/rtc-real-time-clocks-never-sleep>

DS3231.
<https://datasheets.maximintegrated.com/en/ds/DS3231.pdf>

García, V. (2012). “Introducción al I2C Bus”. Consultado en mayo de 2017.
<http://www.diarioelectronicohoy.com/blog/introduccion-al-i2c-bus>

Electroensaimada. “Tutorial Arduino: I2C” Consultado en mayo de 2017.
<http://www.electroensaimada.com/i2c.html>

Arduino. “Wire library”. Consultado en mayo de 2017.
< https://www.arduino.cc/en/Reference/Wire>

Cppreference.
<http://es.cppreference.com/w/>

# 8- Anexos

Se incluye un pequeño índice donde se indican los distintos documentos anexos.

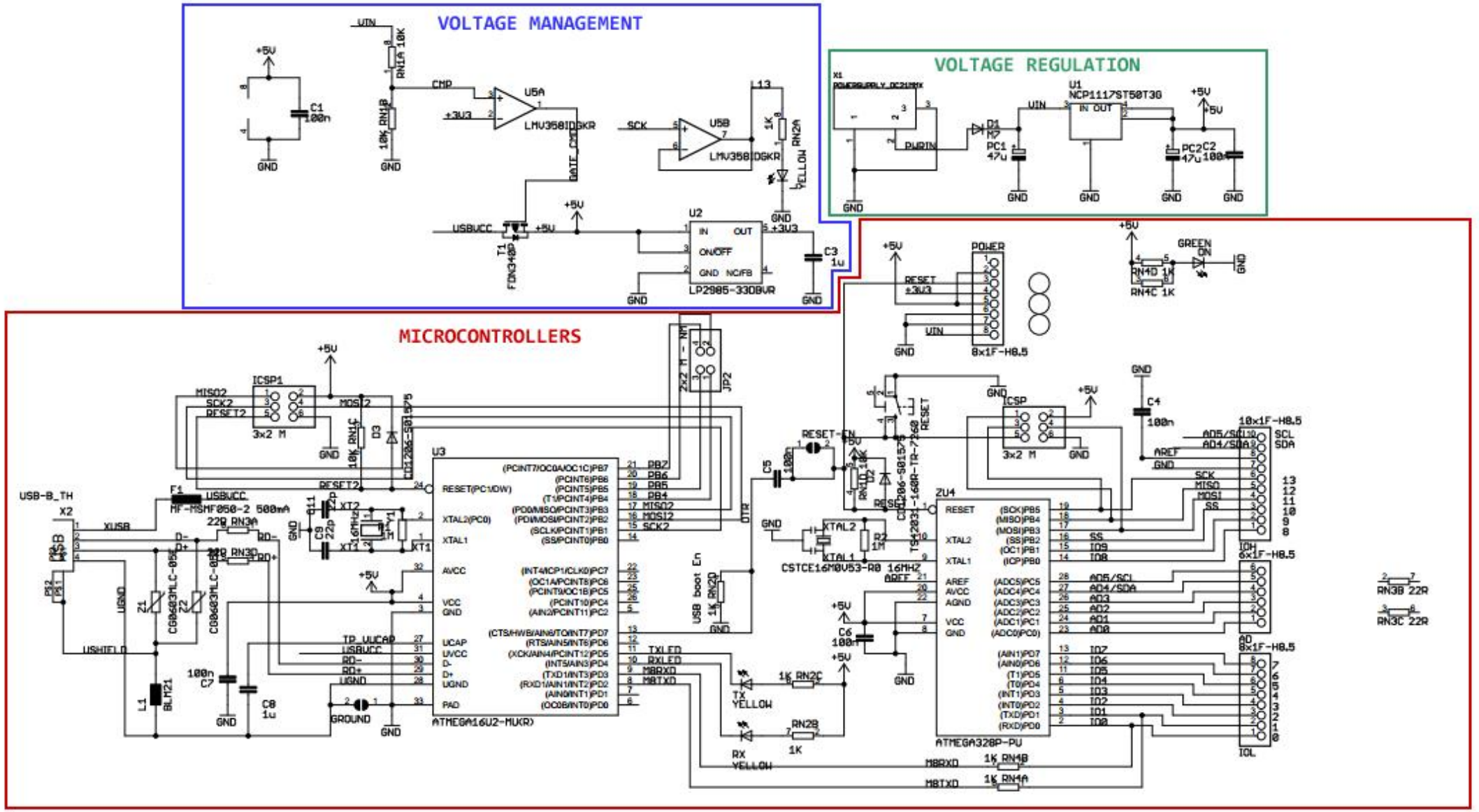
Anexo 1. Esquema eléctrico de Arduino.

Anexo 2. Configuración de pines de Arduino.

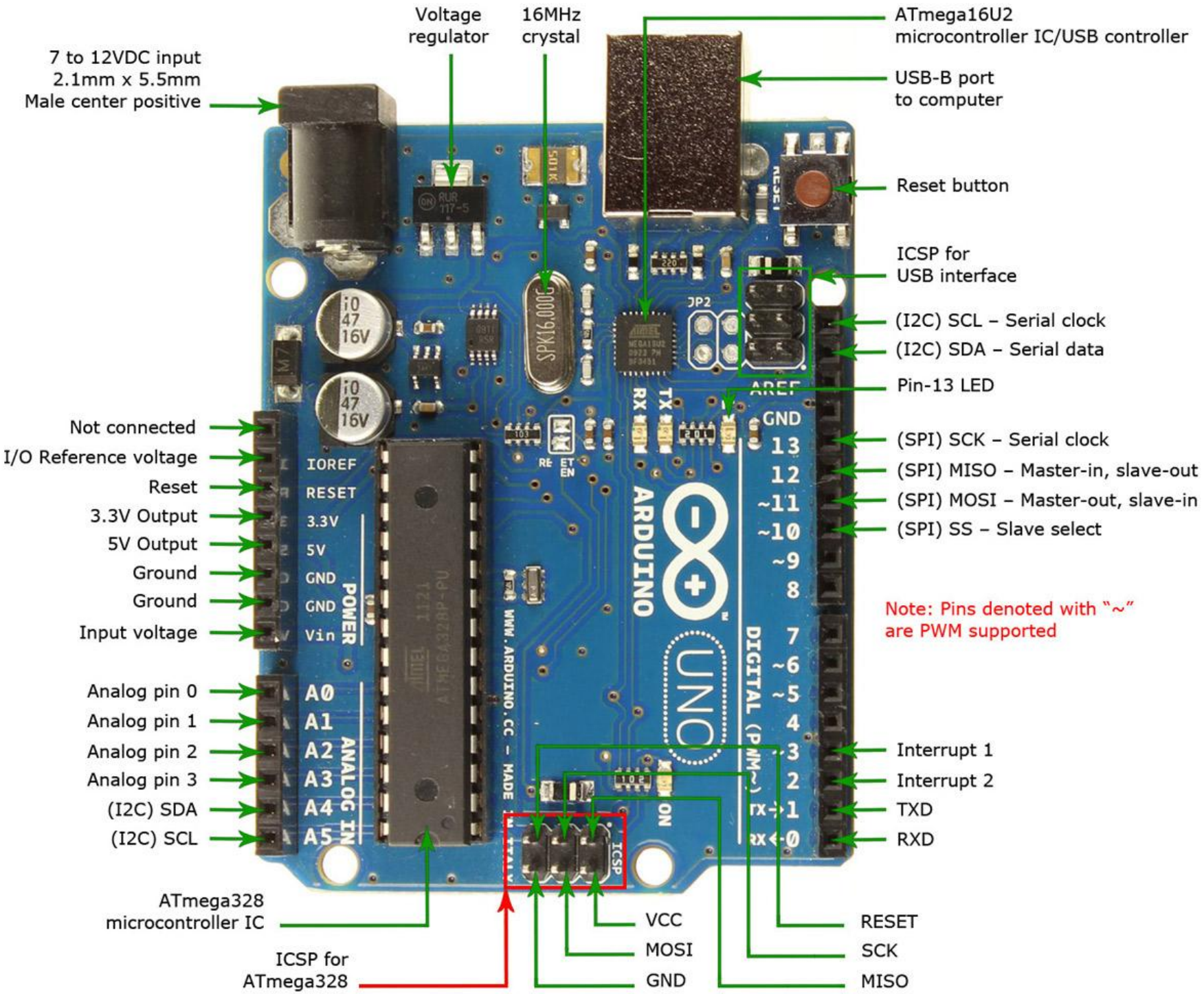
Anexo 3. Datasheet del dispositivo DS3231.

Anexo 4. Manual de usuario de la librería “RTC_lib”.

# Anexo 1
# Esquema eléctrico de Arduino


VOLTAGE MANAGEMENT
VOLTAGE REGULATION
MICROCONTROLLERS
U1
NCP1117ST50T3G
LP2985-33DBVR
LMV358IDGKR
ATMEGA16U2-MUR
ATMEGA328P-PU
CSTCE16M0V53-R0 16MHZ
ICSP1
ICSP
POWER
8x1F-H8.5
10x1F-H8.5
USB-B_TH
GREEN
YELLOW
TX
RX
+5V
GND
+3V3
VIN
USBVCC
SCK
MISO2
MOSI2
RESET2
MBRXD
MBTXD
TXLED
RXLED
SCL
SDA

# Anexo 2
# Configuración de pines de Arduino


7 to 12VDC input
2.1mm x 5.5mm
Male center positive
Voltage regulator
16MHz crystal
ATmega16U2
microcontroller IC/USB controller
USB-B port
to computer
Reset button
ICSP for
USB interface
(I2C) SCL – Serial clock
(I2C) SDA – Serial data
Pin-13 LED
Not connected
I/O Reference voltage
Reset
3.3V Output
5V Output
Ground
Ground
Input voltage
(SPI) SCK – Serial clock
(SPI) MISO – Master-in, slave-out
(SPI) MOSI – Master-out, slave-in
(SPI) SS – Slave select
Note: Pins denoted with "~"
are PWM supported
Analog pin 0
Analog pin 1
Analog pin 2
Analog pin 3
(I2C) SDA
(I2C) SCL
Interrupt 1
Interrupt 2
TXD
RXD
ATmega328
microcontroller IC
ICSP for
ATmega328
VCC
MOSI
GND
RESET
SCK
MISO
IOREF
RESET
3.3V
5V
GND
GND
Vin
POWER
A0
A1
A2
A3
A4
A5
ANALOG IN
ARDUINO
UNO
DIGITAL (PWM~)
AREF
GND
13
12
~11
~10
~9
8
7
~6
~5
4
~3
2
TX→1
RX←0
TX
RX
ON
ICSP
JP2
WWW.ARDUINO.CC – MADE IN ITALY

# Anexo 3

# DataSheet del dispositivo DS3231

# DS3231

# Extremely Accurate I²C-Integrated RTC/TCXO/Crystal

## General Description

The DS3231 is a low-cost, extremely accurate I²C real-time clock (RTC) with an integrated temperature-compensated crystal oscillator (TCXO) and crystal. The device incorporates a battery input, and maintains accurate timekeeping when main power to the device is interrupted. The integration of the crystal resonator enhances the long-term accuracy of the device as well as reduces the piece-part count in a manufacturing line. The DS3231 is available in commercial and industrial temperature ranges, and is offered in a 16-pin, 300-mil SO package.

The RTC maintains seconds, minutes, hours, day, date, month, and year information. The date at the end of the month is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with an $\overline{\text{AM}}$/PM indicator. Two programmable time-of-day alarms and a programmable square-wave output are provided. Address and data are transferred serially through an I²C bidirectional bus.

A precision temperature-compensated voltage reference and comparator circuit monitors the status of $V_{CC}$ to detect power failures, to provide a reset output, and to automatically switch to the backup supply when necessary. Additionally, the $\overline{\text{RST}}$ pin is monitored as a pushbutton input for generating a µP reset.

## Benefits and Features

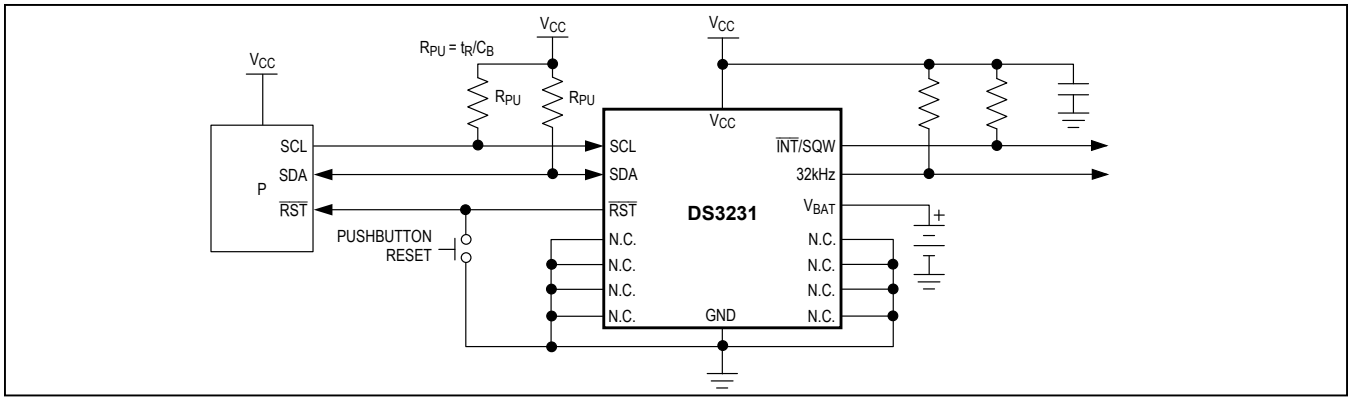
- Highly Accurate RTC Completely Manages All Timekeeping Functions
  - Real-Time Clock Counts Seconds, Minutes, Hours, Date of the Month, Month, Day of the Week, and Year, with Leap-Year Compensation Valid Up to 2100
  - Accuracy ±2ppm from 0°C to +40°C
  - Accuracy ±3.5ppm from -40°C to +85°C
  - Digital Temp Sensor Output: ±3°C Accuracy
  - Register for Aging Trim
  - $\overline{\text{RST}}$ Output/Pushbutton Reset Debounce Input
  - Two Time-of-Day Alarms
  - Programmable Square-Wave Output Signal
- Simple Serial Interface Connects to Most Microcontrollers
  - Fast (400kHz) I²C Interface
- Battery-Backup Input for Continuous Timekeeping
  - Low Power Operation Extends Battery-Backup Run Time
  - 3.3V Operation
- Operating Temperature Ranges: Commercial (0°C to +70°C) and Industrial (-40°C to +85°C)
- Underwriters Laboratories® (UL) Recognized

## Applications

- Servers
- Telematics
- Utility Power Meters
- GPS

***Ordering Information and Pin Configuration appear at end of data sheet.***

## Typical Operating Circuit




*Underwriters Laboratories is a registered certification mark of Underwriters Laboratories Inc.*



maxim integrated™

## Absolute Maximum Ratings

Voltage Range on Any Pin Relative to Ground ....-0.3V to +6.0V
Junction-to-Ambient Thermal Resistance ($\theta_{JA}$) (Note 1)73°C/W
Junction-to-Case Thermal Resistance ($\theta_{JC}$) (Note 1) ....23°C/W
Operating Temperature Range
DS3231S .....0°C to +70°C
DS3231SN..... -40°C to +85°C
Junction Temperature.....+125°C
Storage Temperature Range..... -40°C to +85°C
Lead Temperature (soldering, 10s) .....+260°C
Soldering Temperature (reflow, 2 times max) .....+260°C
(see the *Handling*, *PCB Layout*, *and Assembly* section)

**Note 1:** Package thermal resistances were obtained using the method described in JEDEC specification JESD51-7, using a four-layer board. For detailed information on package thermal considerations, refer to **www.maximintegrated.com/thermal-tutorial**.

*Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. These are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of the specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.*

## Recommended Operating Conditions

($T_A$ = $T_{MIN}$ to $T_{MAX}$, unless otherwise noted.) (Notes 2, 3)

| PARAMETER | SYMBOL | CONDITIONS | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|
| Supply Voltage | $V_{CC}$ | | 2.3 | 3.3 | 5.5 | V |
| | $V_{BAT}$ | | 2.3 | 3.0 | 5.5 | V |
| Logic 1 Input SDA, SCL | $V_{IH}$ | | 0.7 x $V_{CC}$ | | $V_{CC}$ + 0.3 | V |
| Logic 0 Input SDA, SCL | $V_{IL}$ | | -0.3 | | 0.3 x $V_{CC}$ | V |

## Electrical Characteristics

($V_{CC}$ = 2.3V to 5.5V, $V_{CC}$ = Active Supply (see Table 1), $T_A$ = $T_{MIN}$ to $T_{MAX}$, unless otherwise noted.) (Typical values are at $V_{CC}$ = 3.3V, $V_{BAT}$ = 3.0V, and $T_A$ = +25°C, unless otherwise noted.) (Notes 2, 3)

| PARAMETER | SYMBOL | CONDITIONS | | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|---|
| Active Supply Current | $I_{CCA}$ | (Notes 4, 5) | $V_{CC}$ = 3.63V | | | 200 | µA |
| | | | $V_{CC}$ = 5.5V | | | 300 | |
| Standby Supply Current | $I_{CCS}$ | I²C bus inactive, 32kHz output on, SQW output off (Note 5) | $V_{CC}$ = 3.63V | | | 110 | µA |
| | | | $V_{CC}$ = 5.5V | | | 170 | |
| Temperature Conversion Current | $I_{CCSCONV}$ | I²C bus inactive, 32kHz output on, SQW output off | $V_{CC}$ = 3.63V | | | 575 | µA |
| | | | $V_{CC}$ = 5.5V | | | 650 | |
| Power-Fail Voltage | $V_{PF}$ | | | 2.45 | 2.575 | 2.70 | V |
| Logic 0 Output, 32kHz, $\overline{INT}$/SQW, SDA | $V_{OL}$ | $I_{OL}$ = 3mA | | | | 0.4 | V |
| Logic 0 Output, $\overline{RST}$ | $V_{OL}$ | $I_{OL}$ = 1mA | | | | 0.4 | V |
| Output Leakage Current 32kHz, $\overline{INT}$/SQW, SDA | $I_{LO}$ | Output high impedance | | -1 | 0 | +1 | µA |
| Input Leakage SCL | $I_{LI}$ | | | -1 | | +1 | µA |
| $\overline{RST}$ Pin I/O Leakage | $I_{OL}$ | $\overline{RST}$ high impedance (Note 6) | | -200 | | +10 | µA |
| $V_{BAT}$ Leakage Current ($V_{CC}$ Active) | $I_{BATLKG}$ | | | | 25 | 100 | nA |

## Electrical Characteristics (continued)

($V_{CC}$ = 2.3V to 5.5V, $V_{CC}$ = Active Supply (see Table 1), $T_A$ = $T_{MIN}$ to $T_{MAX}$, unless otherwise noted.) (Typical values are at $V_{CC}$ = 3.3V, $V_{BAT}$ = 3.0V, and $T_A$ = +25°C, unless otherwise noted.) (Notes 2, 3)

| PARAMETER | SYMBOL | CONDITIONS | | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|---|
| Output Frequency | $f_{OUT}$ | $V_{CC}$ = 3.3V or $V_{BAT}$ = 3.3V | | | 32.768 | | kHz |
| Frequency Stability vs. Temperature (Commercial) | $\Delta f/f_{OUT}$ | $V_{CC}$ = 3.3V or $V_{BAT}$ = 3.3V, aging offset = 00h | 0°C to +40°C | | | ±2 | ppm |
| | | | >40°C to +70°C | | | ±3.5 | |
| Frequency Stability vs. Temperature (Industrial) | $\Delta f/f_{OUT}$ | $V_{CC}$ = 3.3V or $V_{BAT}$ = 3.3V, aging offset = 00h | -40°C to <0°C | | | ±3.5 | ppm |
| | | | 0°C to +40°C | | | ±2 | |
| | | | >40°C to +85°C | | | ±3.5 | |
| Frequency Stability vs. Voltage | Δf/V | | | | 1 | | ppm/V |
| Trim Register Frequency Sensitivity per LSB | Δf/LSB | Specified at: | -40°C | | 0.7 | | ppm |
| | | | +25°C | | 0.1 | | |
| | | | +70°C | | 0.4 | | |
| | | | +85°C | | 0.8 | | |
| Temperature Accuracy | Temp | $V_{CC}$ = 3.3V or $V_{BAT}$ = 3.3V | | -3 | | +3 | °C |
| Crystal Aging | $\Delta f/f_O$ | After reflow, not production tested | First year | | ±1.0 | | ppm |
| | | | 0–10 years | | ±5.0 | | |

## Electrical Characteristics

(**$V_{CC}$ = 0V, $V_{BAT}$ = 2.3V to 5.5V**, $T_A$ = $T_{MIN}$ to $T_{MAX}$, unless otherwise noted.) (Note 2)

| PARAMETER | SYMBOL | CONDITIONS | | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|---|
| Active Battery Current | $I_{BATA}$ | $\overline{EOSC}$ = 0, BBSQW = 0, SCL = 400kHz (Note 5) | $V_{BAT}$ = 3.63V | | | 70 | µA |
| | | | $V_{BAT}$ = 5.5V | | | 150 | |
| Timekeeping Battery Current | $I_{BATT}$ | $\overline{EOSC}$ = 0, BBSQW = 0, EN32kHz = 1, SCL = SDA = 0V or SCL = SDA = $V_{BAT}$ (Note 5) | $V_{BAT}$ = 3.63V | | 0.84 | 3.0 | µA |
| | | | $V_{BAT}$ = 5.5V | | 1.0 | 3.5 | |
| Temperature Conversion Current | $I_{BATTC}$ | $\overline{EOSC}$ = 0, BBSQW = 0, SCL = SDA = 0V or SCL = SDA = $V_{BAT}$ | $V_{BAT}$ = 3.63V | | | 575 | µA |
| | | | $V_{BAT}$ = 5.5V | | | 650 | |
| Data-Retention Current | $I_{BATTDR}$ | $\overline{EOSC}$ = 1, SCL = SDA = 0V, +25°C | | | | 100 | nA |

## AC Electrical Characteristics

($V_{CC}$ = $V_{CC(MIN)}$ to $V_{CC(MAX)}$ or $V_{BAT}$ = $V_{BAT(MIN)}$ to $V_{BAT(MAX)}$, $V_{BAT}$ > $V_{CC}$, $T_A$ = $T_{MIN}$ to $T_{MAX}$, unless otherwise noted.) (Note 2)

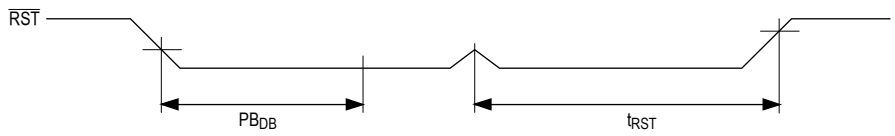
| PARAMETER | SYMBOL | CONDITIONS | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|
| SCL Clock Frequency | $f_{SCL}$ | Fast mode | 100 | | 400 | kHz |
| | | Standard mode | 0 | | 100 | |
| Bus Free Time Between STOP and START Conditions | $t_{BUF}$ | Fast mode | 1.3 | | | µs |
| | | Standard mode | 4.7 | | | |
| Hold Time (Repeated) START Condition (Note 7) | $t_{HD:STA}$ | Fast mode | 0.6 | | | µs |
| | | Standard mode | 4.0 | | | |
| Low Period of SCL Clock | $t_{LOW}$ | Fast mode | 1.3 | | | µs |
| | | Standard mode | 4.7 | | | |
| High Period of SCL Clock | $t_{HIGH}$ | Fast mode | 0.6 | | | µs |
| | | Standard mode | 4.0 | | | |
| Data Hold Time (Notes 8, 9) | $t_{HD:DAT}$ | Fast mode | 0 | | 0.9 | µs |
| | | Standard mode | 0 | | 0.9 | |
| Data Setup Time (Note 10) | $t_{SU:DAT}$ | Fast mode | 100 | | | ns |
| | | Standard mode | 250 | | | |
| START Setup Time | $t_{SU:STA}$ | Fast mode | 0.6 | | | µs |
| | | Standard mode | 4.7 | | | |
| Rise Time of Both SDA and SCL Signals (Note 11) | $t_R$ | Fast mode | 20 + 0.1$C_B$ | | 300 | ns |
| | | Standard mode | | | 1000 | |
| Fall Time of Both SDA and SCL Signals (Note 11) | $t_F$ | Fast mode | 20 + 0.1$C_B$ | | 300 | ns |
| | | Standard mode | | | 300 | |
| Setup Time for STOP Condition | $t_{SU:STO}$ | Fast mode | 0.6 | | | µs |
| | | Standard mode | 4.7 | | | |
| Capacitive Load for Each Bus Line | $C_B$ | (Note 11) | | | 400 | pF |
| Capacitance for SDA, SCL | $C_{I/O}$ | | | 10 | | pF |
| Pulse Width of Spikes That Must Be Suppressed by the Input Filter | $t_{SP}$ | | | 30 | | ns |
| Pushbutton Debounce | $PB_{DB}$ | | | 250 | | ms |
| Reset Active Time | $t_{RST}$ | | | 250 | | ms |
| Oscillator Stop Flag (OSF) Delay | $t_{OSF}$ | (Note 12) | | 100 | | ms |
| Temperature Conversion Time | $t_{CONV}$ | | | 125 | 200 | ms |

## Power-Switch Characteristics

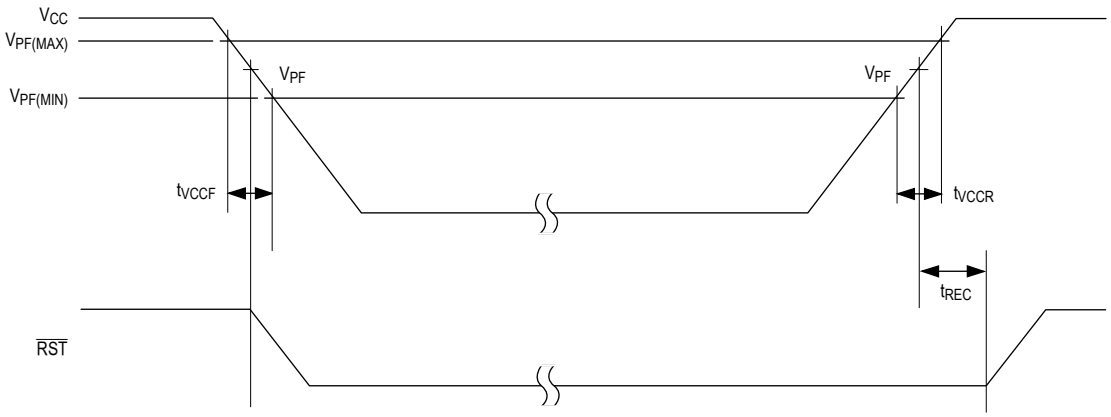
($T_A$ = $T_{MIN}$ to $T_{MAX}$)

| PARAMETER | SYMBOL | CONDITIONS | MIN | TYP | MAX | UNITS |
|---|---|---|---|---|---|---|
| $V_{CC}$ Fall Time; $V_{PF(MAX)}$ to $V_{PF(MIN)}$ | $t_{VCCF}$ | | 300 | | | µs |
| $V_{CC}$ Rise Time; $V_{PF(MIN)}$ to $V_{PF(MAX)}$ | $t_{VCCR}$ | | 0 | | | µs |
| Recovery at Power-Up | $t_{REC}$ | (Note 13) | | 250 | 300 | ms |

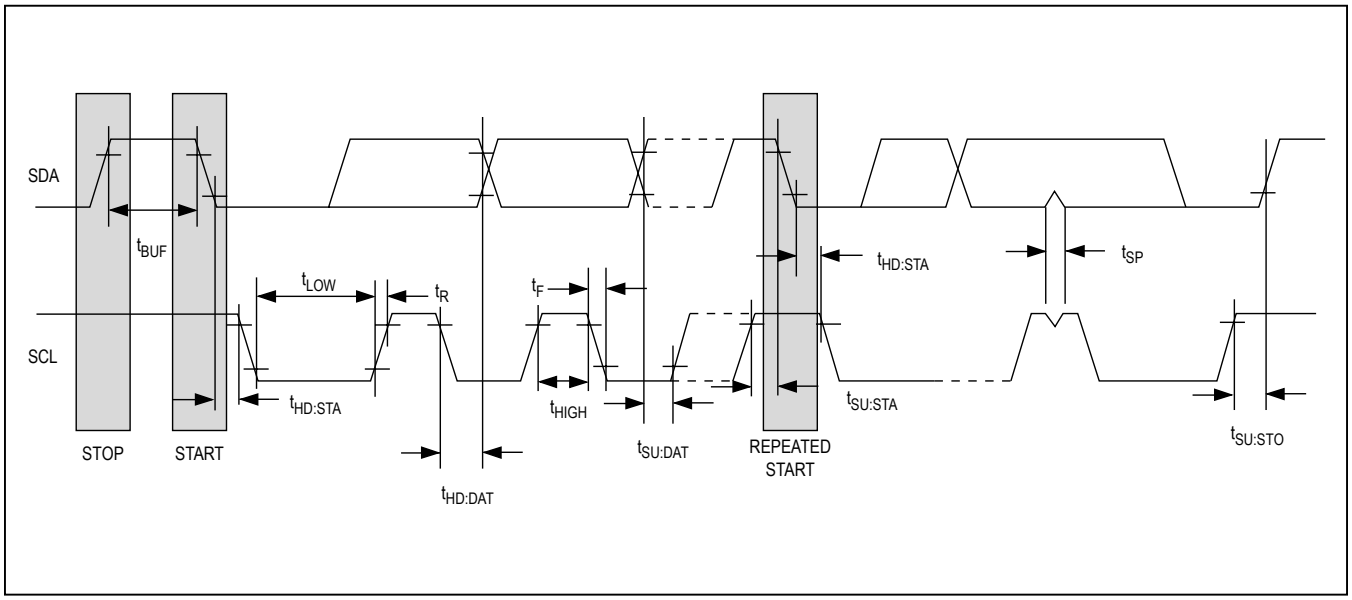
## Pushbutton Reset Timing




## Power-Switch Timing



## Data Transfer on I2C Serial Bus




**WARNING: Negative undershoots below -0.3V while the part is in battery-backed mode may cause loss of data.**

**Note 2:** Limits at -40°C are guaranteed by design and not production tested.

**Note 3:** All voltages are referenced to ground.

**Note 4:** $I_{CCA}$—SCL clocking at max frequency = 400kHz.

**Note 5:** Current is the averaged input current, which includes the temperature conversion current.

**Note 6:** The $\overline{RST}$ pin has an internal 50kΩ (nominal) pullup resistor to $V_{CC}$.

**Note 7:** After this period, the first clock pulse is generated.

**Note 8:** A device must internally provide a hold time of at least 300ns for the SDA signal (referred to the $V_{IH(MIN)}$ of the SCL signal) to bridge the undefined region of the falling edge of SCL.

**Note 9:** The maximum $t_{HD:DAT}$ needs only to be met if the device does not stretch the low period ($t_{LOW}$) of the SCL signal.

**Note 10:** A fast-mode device can be used in a standard-mode system, but the requirement $t_{SU:DAT} \geq 250ns$ must then be met. This is automatically the case if the device does not stretch the low period of the SCL signal. If such a device does stretch the low period of the SCL signal, it must output the next data bit to the SDA line $t_{R(MAX)} + t_{SU:DAT} = 1000 + 250 = 1250ns$ before the SCL line is released.

**Note 11:** $C_B$—total capacitance of one bus line in pF.

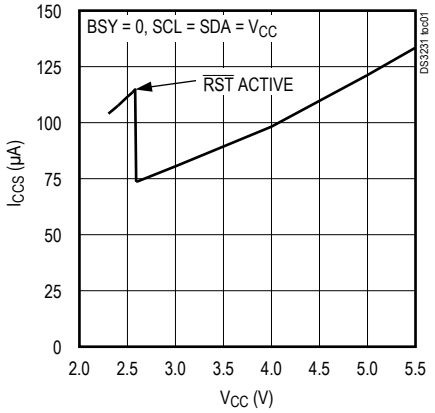
**Note 12:** The parameter $t_{OSF}$ is the period of time the oscillator must be stopped for the OSF flag to be set over the voltage range of $0.0V \leq V_{CC} \leq V_{CC(MAX)}$ and $2.3V \leq V_{BAT} \leq 3.4V$.

**Note 13:** This delay applies only if the oscillator is enabled and running. If the $\overline{EOSC}$ bit is a 1, $t_{REC}$ is bypassed and $\overline{RST}$ immediately goes high. The state of $\overline{RST}$ does not affect the I2C interface, RTC, or TCXO.

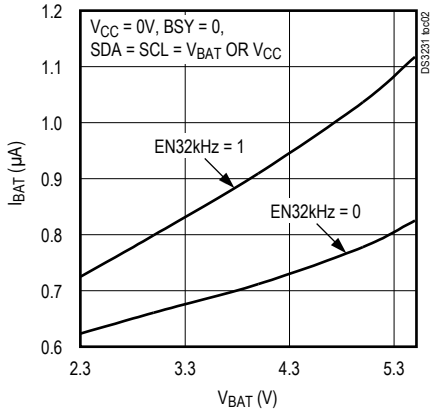
## Typical Operating Characteristics

($V_{CC}$ = +3.3V, $T_A$ = +25°C, unless otherwise noted.)

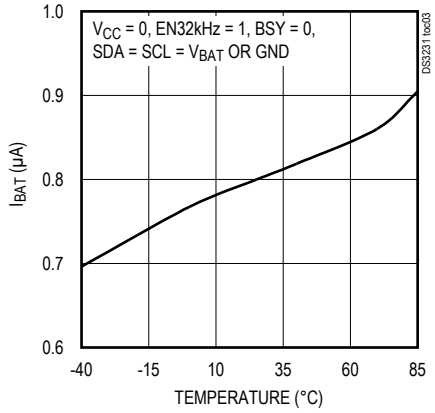
**STANDBY SUPPLY CURRENT vs. SUPPLY VOLTAGE**



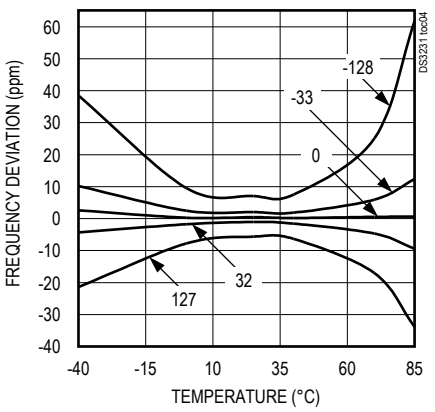

**SUPPLY CURRENT vs. SUPPLY VOLTAGE**



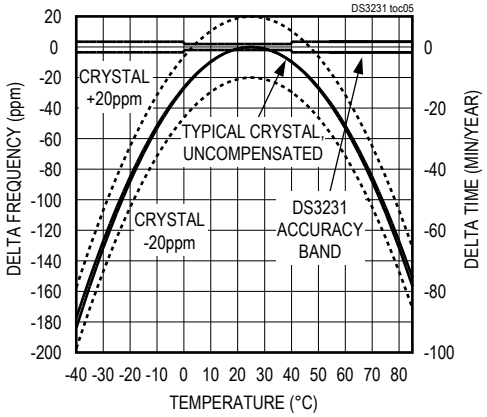

**SUPPLY CURRENT vs. TEMPERATURE**



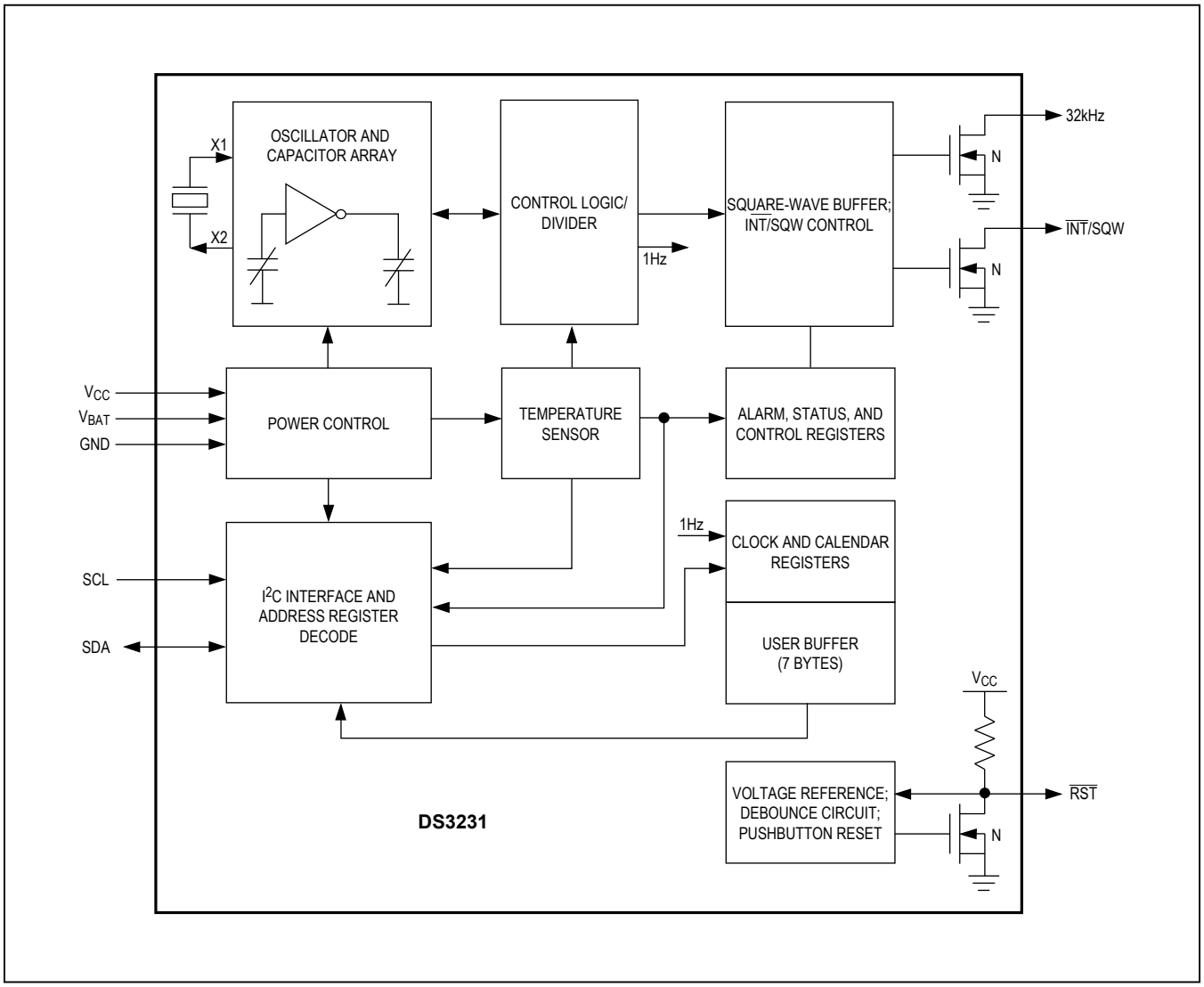

**FREQUENCY DEVIATION vs. TEMPERATURE vs. AGING VALUE**




**DELTA TIME AND FREQUENCY vs. TEMPERATURE**



## Block Diagram



## Pin Description

| PIN | NAME | FUNCTION |
|---|---|---|
| 1 | 32kHz | 32kHz Output. This open-drain pin requires an external pullup resistor. When enabled, the output operates on either power supply. It may be left open if not used. |
| 2 | $V_{CC}$ | DC Power Pin for Primary Power Supply. This pin should be decoupled using a 0.1µF to 1.0µF capacitor. If not used, connect to ground. |
| 3 | $\overline{INT}$/SQW | Active-Low Interrupt or Square-Wave Output. This open-drain pin requires an external pullup resistor connected to a supply at 5.5V or less. This multifunction pin is determined by the state of the INTCN bit in the Control Register (0Eh). When INTCN is set to logic 0, this pin outputs a square wave and its frequency is determined by RS2 and RS1 bits. When INTCN is set to logic 1, then a match between the timekeeping registers and either of the alarm registers activates the $\overline{INT}$/SQW pin (if the alarm is enabled). Because the INTCN bit is set to logic 1 when power is first applied, the pin defaults to an interrupt output with alarms disabled. The pullup voltage can be up to 5.5V, regardless of the voltage on $V_{CC}$. If not used, this pin can be left unconnected. |
| 4 | $\overline{RST}$ | Active-Low Reset. This pin is an open-drain input/output. It indicates the status of $V_{CC}$ relative to the $V_{PF}$ specification. As $V_{CC}$ falls below $V_{PF}$, the $\overline{RST}$ pin is driven low. When $V_{CC}$ exceeds $V_{PF}$, for $t_{RST}$, the $\overline{RST}$ pin is pulled high by the internal pullup resistor. The active-low, open-drain output is combined with a debounced pushbutton input function. This pin can be activated by a pushbutton reset request. It has an internal 50kΩ nominal value pullup resistor to $V_{CC}$. No external pullup resistors should be connected. If the oscillator is disabled, $t_{REC}$ is bypassed and $\overline{RST}$ immediately goes high. |
| 5–12 | N.C. | No Connection. Must be connected to ground. |
| 13 | GND | Ground |
| 14 | $V_{BAT}$ | Backup Power-Supply Input. When using the device with the $V_{BAT}$ input as the primary power source, this pin should be decoupled using a 0.1µF to 1.0µF low-leakage capacitor. When using the device with the $V_{BAT}$ input as the backup power source, the capacitor is not required. If $V_{BAT}$ is not used, connect to ground. The device is UL recognized to ensure against reverse charging when used with a primary lithium battery. Go to www.maximintegrated.com/qa/info/ul. |
| 15 | SDA | Serial Data Input/Output. This pin is the data input/output for the I²C serial interface. This open-drain pin requires an external pullup resistor. The pullup voltage can be up to 5.5V, regardless of the voltage on $V_{CC}$. |
| 16 | SCL | Serial Clock Input. This pin is the clock input for the I²C serial interface and is used to synchronize data movement on the serial interface. Up to 5.5V can be used for this pin, regardless of the voltage on $V_{CC}$. |

## Detailed Description

The DS3231 is a serial RTC driven by a temperature-compensated 32kHz crystal oscillator. The TCXO provides a stable and accurate reference clock, and maintains the RTC to within ±2 minutes per year accuracy from -40°C to +85°C. The TCXO frequency output is available at the 32kHz pin. The RTC is a low-power clock/calendar with two programmable time-of-day alarms and a programmable square-wave output. The $\overline{INT}$/SQW provides either an interrupt signal due to alarm conditions or a square-wave output. The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The date at the end of the month is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with an $\overline{AM}$/PM indicator. The internal registers are accessible though an I²C bus interface.

A temperature-compensated voltage reference and comparator circuit monitors the level of $V_{CC}$ to detect power failures and to automatically switch to the backup supply when necessary. The $\overline{RST}$ pin provides an external pushbutton function and acts as an indicator of a power-fail event.

### Operation

The block diagram shows the main elements of the DS3231. The eight blocks can be grouped into four functional groups: TCXO, power control, pushbutton function, and RTC. Their operations are described separately in the following sections.

### 32kHz TCXO

The temperature sensor, oscillator, and control logic form the TCXO. The controller reads the output of the on-chip temperature sensor and uses a lookup table to determine the capacitance required, adds the aging correction in AGE register, and then sets the capacitance selection registers. New values, including changes to the AGE register, are loaded only when a change in the temperature value occurs, or when a user-initiated temperature conversion is completed. Temperature conversion occurs on initial application of $V_{CC}$ and once every 64 seconds afterwards.

### Power Control

This function is provided by a temperature-compensated voltage reference and a comparator circuit that monitors the $V_{CC}$ level. When $V_{CC}$ is greater than $V_{PF}$, the part is powered by $V_{CC}$. When $V_{CC}$ is less than $V_{PF}$ but greater than $V_{BAT}$, the DS3231 is powered by $V_{CC}$. If $V_{CC}$ is less than $V_{PF}$ and is less than $V_{BAT}$, the device is powered by $V_{BAT}$. See Table 1.

## Table 1. Power Control

| SUPPLY CONDITION | ACTIVE SUPPLY |
|---|---|
| $V_{CC} < V_{PF}$, $V_{CC} < V_{BAT}$ | $V_{BAT}$ |
| $V_{CC} < V_{PF}$, $V_{CC} > V_{BAT}$ | $V_{CC}$ |
| $V_{CC} > V_{PF}$, $V_{CC} < V_{BAT}$ | $V_{CC}$ |
| $V_{CC} > V_{PF}$, $V_{CC} > V_{BAT}$ | $V_{CC}$ |

To preserve the battery, the first time $V_{BAT}$ is applied to the device, the oscillator will not start up until $V_{CC}$ exceeds $V_{PF}$, or until a valid I²C address is written to the part. Typical oscillator startup time is less than one second. Approximately 2 seconds after $V_{CC}$ is applied, or a valid I²C address is written, the device makes a temperature measurement and applies the calculated correction to the oscillator. Once the oscillator is running, it continues to run as long as a valid power source is available ($V_{CC}$ or $V_{BAT}$), and the device continues to measure the temperature and correct the oscillator frequency every 64 seconds.

On the first application of power ($V_{CC}$) or when a valid I²C address is written to the part ($V_{BAT}$), the time and date registers are reset to 01/01/00 01 00:00:00 (DD/MM/YY DOW HH:MM:SS).

## $V_{BAT}$ Operation

There are several modes of operation that affect the amount of $V_{BAT}$ current that is drawn. While the device is powered by $V_{BAT}$ and the serial interface is active, active battery current, $I_{BATA}$, is drawn. When the serial interface is inactive, timekeeping current ($I_{BATT}$), which includes the averaged temperature conversion current, $I_{BATTC}$, is used (refer to Application Note 3644: *Power Considerations for Accurate Real-Time Clocks* for details). Temperature conversion current, $I_{BATTC}$, is specified since the system must be able to support the periodic higher current pulse and still maintain a valid voltage level. Data retention current, $I_{BATTDR}$, is the current drawn by the part when the oscillator is stopped ($\overline{EOSC}$ = 1). This mode can be used to minimize battery requirements for times when maintaining time and date information is not necessary, e.g., while the end system is waiting to be shipped to a customer.

### Pushbutton Reset Function

The DS3231 provides for a pushbutton switch to be connected to the $\overline{RST}$ output pin. When the DS3231 is not in a reset cycle, it continuously monitors the $\overline{RST}$ signal for a low going edge. If an edge transition is detected, the DS3231 debounces the switch by pulling the $\overline{RST}$ low. After the internal timer has expired ($PB_{DB}$), the DS3231 continues to monitor the $\overline{RST}$ line. If the line is still low, the DS3231 continuously monitors the line looking for a rising edge. Upon detecting release, the DS3231 forces the $\overline{RST}$ pin low and holds it low for $t_{RST}$.

$\overline{RST}$ is also used to indicate a power-fail condition. When $V_{CC}$ is lower than $V_{PF}$, an internal power-fail signal is generated, which forces the $\overline{RST}$ pin low. When $V_{CC}$ returns to a level above $V_{PF}$, the $\overline{RST}$ pin is held low for approximately 250ms ($t_{REC}$) to allow the power supply to stabilize. If the oscillator is not running (see the *Power Control* section) when $V_{CC}$ is applied, $t_{REC}$ is bypassed and $\overline{RST}$ immediately goes high. Assertion of the $\overline{RST}$ output, whether by pushbutton or power-fail detection, does not affect the internal operation of the DS3231.

### Real-Time Clock

With the clock source from the TCXO, the RTC provides seconds, minutes, hours, day, date, month, and year information. The date at the end of the month is automatically adjusted for months with fewer than 31 days, including corrections for leap year. The clock operates in either the 24-hour or 12-hour format with an $\overline{AM}$/PM indicator.

The clock provides two programmable time-of-day alarms and a programmable square-wave output. The $\overline{INT}$/SQW pin either generates an interrupt due to alarm condition or outputs a square-wave signal and the selection is controlled by the bit INTCN.

| ADDRESS | BIT 7 MSB | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 LSB | FUNCTION | RANGE |
|---|---|---|---|---|---|---|---|---|---|---|
| 00h | 0 | 10 Seconds | | | Seconds | | | | Seconds | 00–59 |
| 01h | 0 | 10 Minutes | | | Minutes | | | | Minutes | 00–59 |
| 02h | 0 | 12/$\overline{24}$ | $\overline{AM}$/PM<br>20 Hour | 10 Hour | Hour | | | | Hours | 1–12 + $\overline{AM}$/PM<br>00–23 |
| 03h | 0 | 0 | 0 | 0 | 0 | Day | | | Day | 1–7 |
| 04h | 0 | 0 | 10 Date | | Date | | | | Date | 01–31 |
| 05h | Century | 0 | 0 | 10 Month | Month | | | | Month/ Century | 01–12 + Century |
| 06h | 10 Year | | | | Year | | | | Year | 00–99 |
| 07h | A1M1 | 10 Seconds | | | Seconds | | | | Alarm 1 Seconds | 00–59 |
| 08h | A1M2 | 10 Minutes | | | Minutes | | | | Alarm 1 Minutes | 00–59 |
| 09h | A1M3 | 12/$\overline{24}$ | $\overline{AM}$/PM<br>20 Hour | 10 Hour | Hour | | | | Alarm 1 Hours | 1–12 + $\overline{AM}$/PM<br>00–23 |
| 0Ah | A1M4 | DY/$\overline{DT}$ | 10 Date | | Day | | | | Alarm 1 Day | 1–7 |
| | | | | | Date | | | | Alarm 1 Date | 1–31 |
| 0Bh | A2M2 | 10 Minutes | | | Minutes | | | | Alarm 2 Minutes | 00–59 |
| 0Ch | A2M3 | 12/$\overline{24}$ | $\overline{AM}$/PM<br>20 Hour | 10 Hour | Hour | | | | Alarm 2 Hours | 1–12 + $\overline{AM}$/PM<br>00–23 |
| 0Dh | A2M4 | DY/$\overline{DT}$ | 10 Date | | Day | | | | Alarm 2 Day | 1–7 |
| | | | | | Date | | | | Alarm 2 Date | 1–31 |
| 0Eh | $\overline{EOSC}$ | BBSQW | CONV | RS2 | RS1 | INTCN | A2IE | A1IE | Control | — |
| 0Fh | OSF | 0 | 0 | 0 | EN32kHz | BSY | A2F | A1F | Control/Status | — |
| 10h | SIGN | DATA | DATA | DATA | DATA | DATA | DATA | DATA | Aging Offset | — |
| 11h | SIGN | DATA | DATA | DATA | DATA | DATA | DATA | DATA | MSB of Temp | — |
| 12h | DATA | DATA | 0 | 0 | 0 | 0 | 0 | 0 | LSB of Temp | — |

*Figure 1. Timekeeping Registers*

***Note:*** *Unless otherwise specified, the registers' state is not defined when power is first applied.*

## Address Map

Figure 1 shows the address map for the DS3231 timekeeping registers. During a multibyte access, when the address pointer reaches the end of the register space (12h), it wraps around to location 00h. On an I²C START or address pointer incrementing to location 00h, the current time is transferred to a second set of registers. The time information is read from these secondary registers, while the clock may continue to run. This eliminates the need to reread the registers in case the main registers update during a read.

## I²C Interface

The I²C interface is accessible whenever either $V_{CC}$ or $V_{BAT}$ is at a valid level. If a microcontroller connected to the DS3231 resets because of a loss of $V_{CC}$ or other event, it is possible that the microcontroller and DS3231 I²C communications could become unsynchronized, e.g., the microcontroller resets while reading data from the DS3231. When the microcontroller resets, the DS3231 I²C interface may be placed into a known state by toggling SCL until SDA is observed to be at a high level. At that point the microcontroller should pull SDA low while SCL is high, generating a START condition.

## Clock and Calendar

The time and calendar information is obtained by reading the appropriate register bytes. Figure 1 illustrates the RTC registers. The time and calendar data are set or initialized by writing the appropriate register bytes. The contents of the time and calendar registers are in the binary-coded

decimal (BCD) format. The DS3231 can be run in either 12-hour or 24-hour mode. Bit 6 of the hours register is defined as the 12- or 24-hour mode select bit. When high, the 12-hour mode is selected. In the 12-hour mode, bit 5 is the AM/PM bit with logic-high being PM. In the 24-hour mode, bit 5 is the 20-hour bit (20–23 hours). The century bit (bit 7 of the month register) is toggled when the years register overflows from 99 to 00.

The day-of-week register increments at midnight. Values that correspond to the day of week are user-defined but must be sequential (i.e., if 1 equals Sunday, then 2 equals Monday, and so on). Illogical time and date entries result in undefined operation.

When reading or writing the time and date registers, secondary (user) buffers are used to prevent errors when the internal registers update. When reading the time and date registers, the user buffers are synchronized to the internal registers on any START and when the register pointer rolls over to zero. The time information is read from these secondary registers, while the clock continues to run. This eliminates the need to reread the registers in case the main registers update during a read.

The countdown chain is reset whenever the seconds register is written. Write transfers occur on the acknowledge from the DS3231. Once the countdown chain is reset, to avoid rollover issues the remaining time and date registers must be written within 1 second. The 1Hz square-wave output, if enabled, transitions high 500ms after the seconds data transfer, provided the oscillator is already running.

## Alarms

The DS3231 contains two time-of-day/date alarms. Alarm 1 can be set by writing to registers 07h to 0Ah. Alarm 2 can be set by writing to registers 0Bh to 0Dh. The alarms can be programmed (by the alarm enable and INTCN bits of the control register) to activate the INT/SQW output on an alarm match condition. Bit 7 of each of the time-of-day/date alarm registers are mask bits (Table 2). When all the mask bits for each alarm are logic 0, an alarm only occurs when the values in the timekeeping registers match the corresponding values stored in the time-of-day/date alarm registers. The alarms can also be programmed to repeat every second, minute, hour, day, or date. Table 2 shows the possible settings. Configurations not listed in the table will result in illogical operation.

The DY/DT bits (bit 6 of the alarm day/date registers) control whether the alarm value stored in bits 0 to 5 of that register reflects the day of the week or the date of the month. If DY/DT is written to logic 0, the alarm will be the result of a match with date of the month. If DY/DT is written to logic 1, the alarm will be the result of a match with day of the week.

When the RTC register values match alarm register settings, the corresponding Alarm Flag ‘A1F’ or ‘A2F’ bit is set to logic 1. If the corresponding Alarm Interrupt Enable ‘A1IE’ or ‘A2IE’ is also set to logic 1 and the INTCN bit is set to logic 1, the alarm condition will activate the INT/SQW signal. The match is tested on the once-per-second update of the time and date registers.

## Table 2. Alarm Mask Bits

| DY/DT | ALARM 1 REGISTER MASK BITS (BIT 7) | | | | ALARM RATE |
|---|---|---|---|---|---|
| | A1M4 | A1M3 | A1M2 | A1M1 | |
| X | 1 | 1 | 1 | 1 | Alarm once per second |
| X | 1 | 1 | 1 | 0 | Alarm when seconds match |
| X | 1 | 1 | 0 | 0 | Alarm when minutes and seconds match |
| X | 1 | 0 | 0 | 0 | Alarm when hours, minutes, and seconds match |
| 0 | 0 | 0 | 0 | 0 | Alarm when date, hours, minutes, and seconds match |
| 1 | 0 | 0 | 0 | 0 | Alarm when day, hours, minutes, and seconds match |

| DY/DT | ALARM 2 REGISTER MASK BITS (BIT 7) | | | ALARM RATE |
|---|---|---|---|---|
| | A2M4 | A2M3 | A2M2 | |
| X | 1 | 1 | 1 | Alarm once per minute (00 seconds of every minute) |
| X | 1 | 1 | 0 | Alarm when minutes match |
| X | 1 | 0 | 0 | Alarm when hours and minutes match |
| 0 | 0 | 0 | 0 | Alarm when date, hours, and minutes match |
| 1 | 0 | 0 | 0 | Alarm when day, hours, and minutes match |

### Control Register (0Eh)

| | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|---|
| NAME: | $\overline{EOSC}$ | BBSQW | CONV | RS2 | RS1 | INTCN | A2IE | A1IE |
| POR: | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

## Special-Purpose Registers

The DS3231 has two additional registers (control and status) that control the real-time clock, alarms, and square-wave output.

### Control Register (0Eh)

**Bit 7: Enable Oscillator ($\overline{EOSC}$).** When set to logic 0, the oscillator is started. When set to logic 1, the oscillator is stopped when the DS3231 switches to $V_{BAT}$. This bit is clear (logic 0) when power is first applied. When the DS3231 is powered by $V_{CC}$, the oscillator is always on regardless of the status of the $\overline{EOSC}$ bit. When $\overline{EOSC}$ is disabled, all register data is static.

**Bit 6: Battery-Backed Square-Wave Enable (BBSQW).** When set to logic 1 with INTCN = 0 and $V_{CC} < V_{PF}$, this bit enables the square wave. When BBSQW is logic 0, the $\overline{INT}$/SQW pin goes high impedance when $V_{CC} < V_{PF}$. This bit is disabled (logic 0) when power is first applied.

**Bit 5: Convert Temperature (CONV).** Setting this bit to 1 forces the temperature sensor to convert the temperature into digital code and execute the TCXO algorithm to update the capacitance array to the oscillator. This can only happen when a conversion is not already in progress. The user should check the status bit BSY before forcing the controller to start a new TCXO execution. A user-initiated temperature conversion does not affect the internal 64-second update cycle.

A user-initiated temperature conversion does not affect the BSY bit for approximately 2ms. The CONV bit remains at a 1 from the time it is written until the conversion is finished, at which time both CONV and BSY go to 0. The CONV bit should be used when monitoring the status of a user-initiated conversion.

**Bits 4 and 3: Rate Select (RS2 and RS1).** These bits control the frequency of the square-wave output when the square wave has been enabled. The following table shows the square-wave frequencies that can be selected with the RS bits. These bits are both set to logic 1 (8.192kHz) when power is first applied.

#### SQUARE-WAVE OUTPUT FREQUENCY

| RS2 | RS1 | SQUARE-WAVE OUTPUT FREQUENCY |
|---|---|---|
| 0 | 0 | 1Hz |
| 0 | 1 | 1.024kHz |
| 1 | 0 | 4.096kHz |
| 1 | 1 | 8.192kHz |

**Bit 2: Interrupt Control (INTCN).** This bit controls the $\overline{INT}$/SQW signal. When the INTCN bit is set to logic 0, a square wave is output on the $\overline{INT}$/SQW pin. When the INTCN bit is set to logic 1, then a match between the timekeeping registers and either of the alarm registers activates the $\overline{INT}$/SQW output (if the alarm is also enabled). The corresponding alarm flag is always set regardless of the state of the INTCN bit. The INTCN bit is set to logic 1 when power is first applied.

**Bit 1: Alarm 2 Interrupt Enable (A2IE).** When set to logic 1, this bit permits the alarm 2 flag (A2F) bit in the status register to assert $\overline{INT}$/SQW (when INTCN = 1). When the A2IE bit is set to logic 0 or INTCN is set to logic 0, the A2F bit does not initiate an interrupt signal. The A2IE bit is disabled (logic 0) when power is first applied.

**Bit 0: Alarm 1 Interrupt Enable (A1IE).** When set to logic 1, this bit permits the alarm 1 flag (A1F) bit in the status register to assert $\overline{INT}$/SQW (when INTCN = 1). When the A1IE bit is set to logic 0 or INTCN is set to logic 0, the A1F bit does not initiate the $\overline{INT}$/SQW signal. The A1IE bit is disabled (logic 0) when power is first applied.

### Status Register (0Fh)

| | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|---|
| **NAME:** | OSF | 0 | 0 | 0 | EN32kHz | BSY | A2F | A1F |
| **POR:** | 1 | 0 | 0 | 0 | 1 | X | X | X |

### Status Register (0Fh)

**Bit 7: Oscillator Stop Flag (OSF).** A logic 1 in this bit indicates that the oscillator either is stopped or was stopped for some period and may be used to judge the validity of the timekeeping data. This bit is set to logic 1 any time that the oscillator stops. The following are examples of conditions that can cause the OSF bit to be set:

1) The first time power is applied.
2) The voltages present on both $V_{CC}$ and $V_{BAT}$ are insufficient to support oscillation.
3) The $\overline{EOSC}$ bit is turned off in battery-backed mode.
4) External influences on the crystal (i.e., noise, leakage, etc.).

This bit remains at logic 1 until written to logic 0.

**Bit 3: Enable 32kHz Output (EN32kHz).** This bit controls the status of the 32kHz pin. When set to logic 1, the 32kHz pin is enabled and outputs a 32.768kHz square-wave signal. When set to logic 0, the 32kHz pin goes to a high-impedance state. The initial power-up state of this bit is logic 1, and a 32.768kHz square-wave signal appears at the 32kHz pin after a power source is applied to the DS3231 (if the oscillator is running).

**Bit 2: Busy (BSY).** This bit indicates the device is busy executing TCXO functions. It goes to logic 1 when the conversion signal to the temperature sensor is asserted and then is cleared when the device is in the 1-minute idle state.

**Bit 1: Alarm 2 Flag (A2F).** A logic 1 in the alarm 2 flag bit indicates that the time matched the alarm 2 registers. If the A2IE bit is logic 1 and the INTCN bit is set to logic 1, the $\overline{INT}$/SQW pin is also asserted. A2F is cleared when written to logic 0. This bit can only be written to logic 0. Attempting to write to logic 1 leaves the value unchanged.

**Bit 0: Alarm 1 Flag (A1F).** A logic 1 in the alarm 1 flag bit indicates that the time matched the alarm 1 registers. If the A1IE bit is logic 1 and the INTCN bit is set to logic 1, the $\overline{INT}$/SQW pin is also asserted. A1F is cleared when written to logic 0. This bit can only be written to logic 0. Attempting to write to logic 1 leaves the value unchanged.

## Aging Offset

The aging offset register takes a user-provided value to add to or subtract from the codes in the capacitance array registers. The code is encoded in two's complement, with bit 7 representing the sign bit. One LSB represents one small capacitor to be switched in or out of the capacitance array at the crystal pins. The aging offset register capacitance value is added or subtracted from the capacitance value that the device calculates for each temperature compensation. The offset register is added to the capacitance array during a normal temperature conversion, if the temperature changes from the previous conversion, or during a manual user conversion (setting the CONV bit). To see the effects of the aging register on the 32kHz output frequency immediately, a manual conversion should be started after each aging register change.

Positive aging values add capacitance to the array, slowing the oscillator frequency. Negative values remove capacitance from the array, increasing the oscillator frequency.

The change in ppm per LSB is different at different temperatures. The frequency vs. temperature curve is shifted by the values used in this register. At +25°C, one LSB typically provides about 0.1ppm change in frequency.

Use of the aging register is not needed to achieve the accuracy as defined in the EC tables, but could be used to help compensate for aging at a given temperature. See the *Typical Operating Characteristics* section for a graph showing the effect of the register on accuracy over temperature.

### Aging Offset (10h)

| | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|---|
| **NAME:** | Sign | Data | Data | Data | Data | Data | Data | Data |
| **POR:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Temperature Register (Upper Byte) (11h)

| | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|---|
| **NAME:** | Sign | Data | Data | Data | Data | Data | Data | Data |
| **POR:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### Temperature Register (Lower Byte) (12h)

| | BIT 7 | BIT 6 | BIT 5 | BIT 4 | BIT 3 | BIT 2 | BIT 1 | BIT 0 |
|---|---|---|---|---|---|---|---|---|
| **NAME:** | Data | Data | 0 | 0 | 0 | 0 | 0 | 0 |
| **POR:** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Temperature Registers (11h–12h)

Temperature is represented as a 10-bit code with a resolution of 0.25°C and is accessible at location 11h and 12h. The temperature is encoded in two's complement format. The upper 8 bits, the integer portion, are at location 11h and the lower 2 bits, the fractional portion, are in the upper nibble at location 12h. For example, 00011001 01b = +25.25°C. Upon power reset, the registers are set to a default temperature of 0°C and the controller starts a temperature conversion. The temperature is read on initial application of $V_{CC}$ or I2C access on $V_{BAT}$ and once every 64 seconds afterwards. The temperature registers are updated after each user-initiated conversion and on every 64-second conversion. The temperature registers are read-only.

## I2C Serial Data Bus

The DS3231 supports a bidirectional I2C bus and data transmission protocol. A device that sends data onto the bus is defined as a transmitter and a device receiving data is defined as a receiver. The device that controls the message is called a master. The devices that are controlled by the master are 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 DS3231 operates as a slave on the I2C bus. Connections to the bus are made through the SCL input and open-drain SDA I/O lines. Within the bus specifications, a standard mode (100kHz maximum clock rate) and a fast mode (400kHz maximum clock rate) are defined. The DS3231 works in both modes.

The following bus protocol has been defined (Figure 2):

- Data transfer may 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 are interpreted as control signals.

Accordingly, the following bus conditions have been defined:

**Bus not busy:** Both data and clock lines remain high.

**START data transfer:** A change in the state of the data line from high to low, while the clock line is high, defines a START condition.

**STOP data transfer:** A change in the state of the data line from low to high, while the clock line is high, defines a STOP condition.

**Data valid:** The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the high period of the clock signal. The data on the line must be changed during the low period of the clock signal. There is one clock pulse per bit of data.

Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes transferred between the START and the STOP conditions is not limited, and is determined by the master device. The information is transferred byte-wise and each receiver acknowledges with a ninth bit.

**Acknowledge:** Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse, which is associated with this acknowledge bit.

A device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is stable low during the high period of the acknowledge-related clock pulse. Of course, setup and hold times must be taken into account. A master must signal an end of data to the slave by not generat-

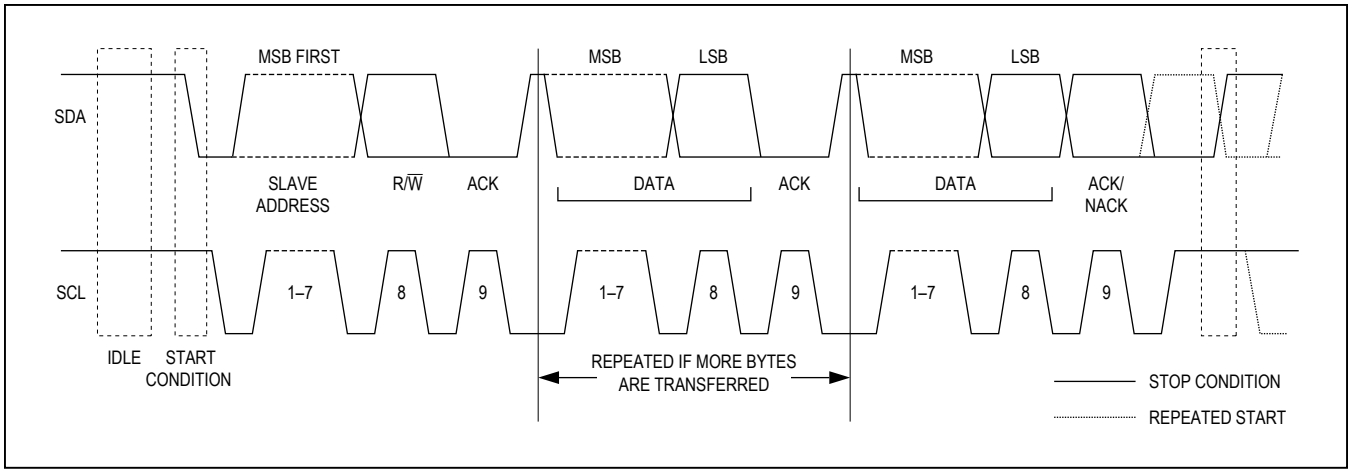



*Figure 2. I2C Data Transfer Overview*

ing an acknowledge bit on the last byte that has been clocked out of the slave. In this case, the slave must leave the data line high to enable the master to generate the STOP condition.

Figures 3 and 4 detail how data transfer is accomplished on the I2C bus. Depending upon the state of the R/$\overline{W}$ bit, two types of data transfer are possible:

**Data transfer from a master transmitter to a slave receiver.** The first byte transmitted by the master is the slave address. Next follows a number of data bytes. The slave returns an acknowledge bit after each received byte. Data is transferred with the most significant bit (MSB) first.

**Data transfer from a slave transmitter to a master receiver.** The first byte (the slave address) is transmitted by the master. The slave then returns an acknowledge bit. Next follows a number of data bytes transmitted by the slave to the master. The master returns an acknowledge bit after all received bytes other than the

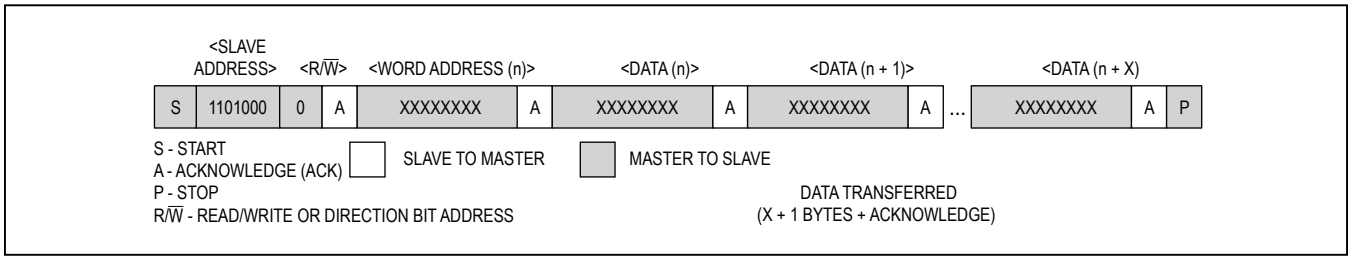



*Figure 3. Data Write—Slave Receiver Mode*

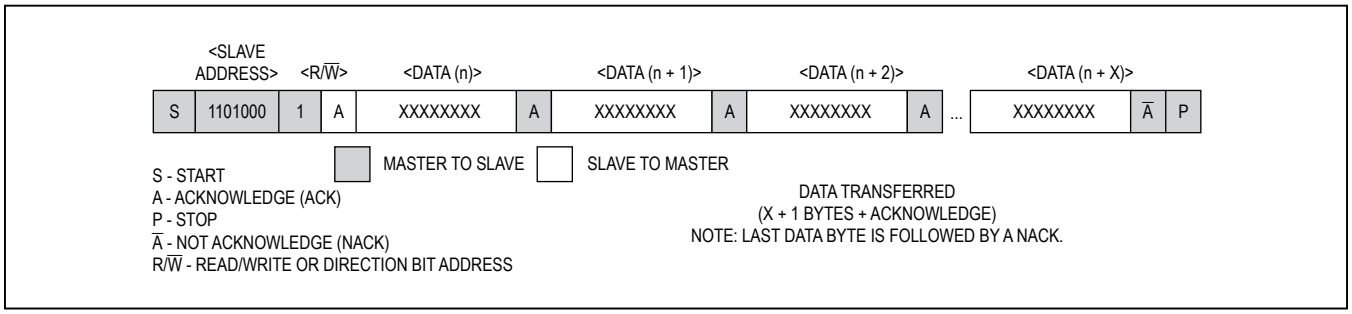



*Figure 4. Data Read—Slave Transmitter Mode*

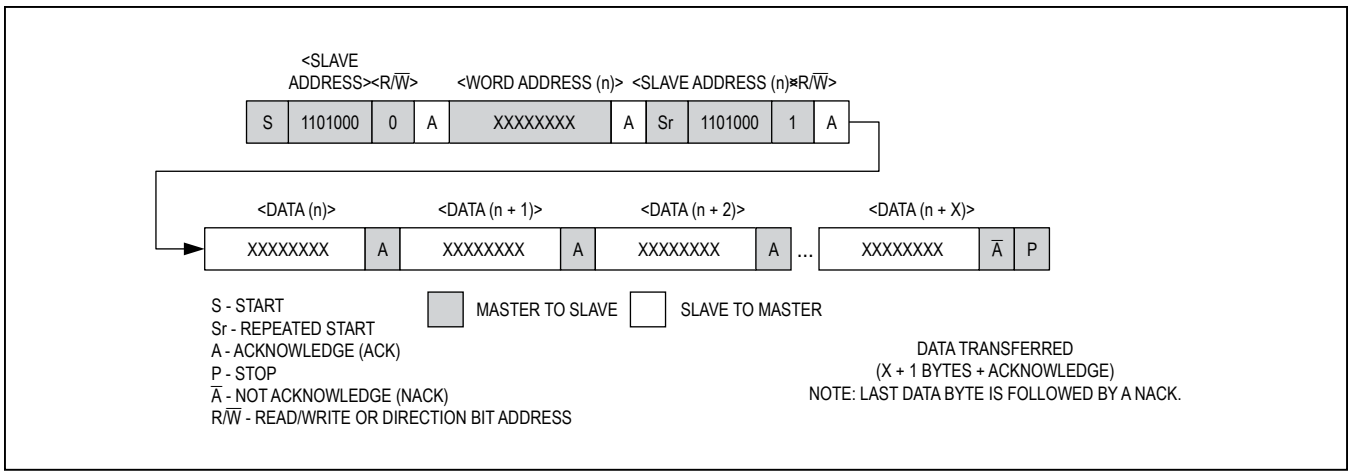



*Figure 5. Data Write/Read (Write Pointer, Then Read)—Slave Receive and Transmit*

last byte. At the end of the last received byte, a not acknowledge is returned.

The master device generates all the serial clock pulses and the START and STOP conditions. A transfer is ended with a STOP condition or with a repeated START condition. Since a repeated START condition is also the beginning of the next serial transfer, the bus will not be released. Data is transferred with the most significant bit (MSB) first.

The DS3231 can operate in the following two modes:

**Slave receiver mode (DS3231 write mode):** Serial data and clock are received through SDA and SCL. After each byte is received, an acknowledge bit is transmitted. START and STOP conditions are recognized as the beginning and end of a serial transfer. Address recognition is performed by hardware after reception of the slave address and direction bit. The slave address byte is the first byte received after the master generates the START condition. The slave address byte contains the 7-bit DS3231 address, which is 1101000, followed by the direction bit (R/$\overline{W}$), which is 0 for a write. After receiving and decoding the slave address byte, the DS3231 outputs an acknowledge on SDA. After the DS3231 acknowledges the slave address + write bit, the master transmits a word address to the DS3231. This sets the register pointer on the DS3231, with the DS3231 acknowledging the transfer. The master may then transmit zero or more bytes of data, with the DS3231 acknowledging each byte received. The register pointer increments after each data byte is transferred. The master generates a STOP condition to terminate the data write.

**Slave transmitter mode (DS3231 read mode):** The first byte is received and handled as in the slave receiver mode. However, in this mode, the direction bit indicates that the transfer direction is reversed. Serial data is transmitted on SDA by the DS3231 while the serial clock is input on SCL. START and STOP conditions are recognized as the beginning and end of a serial transfer. Address recognition is performed by hardware after reception of the slave address and direction bit. The slave address byte is the first byte received after the master generates a START condition. The slave address byte contains the 7-bit DS3231 address, which is 1101000, followed by the direction bit (R/$\overline{W}$), which is 1 for a read. After receiving and decoding the slave address byte, the DS3231 outputs an acknowledge on SDA. The DS3231 then begins to transmit data starting with the register address pointed to by the register pointer. If the register pointer is not written to before the initiation of a read mode, the first address that is read is the last one stored in the register pointer. The DS3231 must receive a not acknowledge to end a read.

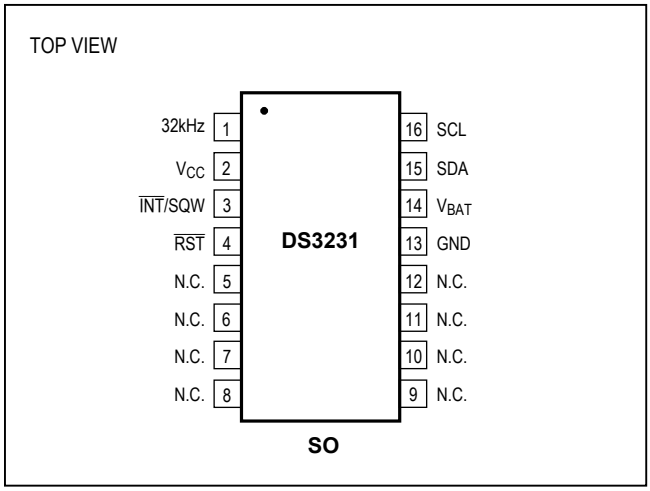
## Handling, PCB Layout, and Assembly

The DS3231 package contains a quartz tuning-fork crystal. Pick-and-place equipment can be used, but precautions should be taken to ensure that excessive shocks are avoided. Ultrasonic cleaning should be avoided to prevent damage to the crystal.

Avoid running signal traces under the package, unless a ground plane is placed between the package and the signal line. All N.C. (no connect) pins must be connected to ground.

Moisture-sensitive packages are shipped from the factory dry packed. Handling instructions listed on the package label must be followed to prevent damage during reflow. Refer to the IPC/JEDEC J-STD-020 standard for moisture-sensitive device (MSD) classifications and reflow profiles. Exposure to reflow is limited to 2 times maximum.

## Pin Configuration




## Chip Information

SUBSTRATE CONNECTED TO GROUND

PROCESS: CMOS

## Ordering Information

| PART | TEMP RANGE | PIN-PACKAGE |
|---|---|---|
| DS3231S# | 0°C to +70°C | 16 SO |
| DS3231SN# | -40°C to +85°C | 16 SO |

*#Denotes an RoHS-compliant device that may include lead (Pb) that is exempt under RoHS requirements. The lead finish is JESD97 category e3, and is compatible with both lead-based and lead-free soldering processes. A "#" anywhere on the top mark denotes an RoHS-compliant device.*

## Package Information

For the latest package outline information and land patterns (footprints), go to **www.maximintegrated.com/packages**. Note that a "+", "#", or "-" in the package code indicates RoHS status only. Package drawings may show a different suffix character, but the drawing pertains to the package regardless of RoHS status.

| PACKAGE TYPE | PACKAGE CODE | OUTLINE NO. | LAND PATTERN NO. |
|---|---|---|---|
| 16 SO | W16#H2 | 21-0042 | 90-0107 |

## Revision History

| REVISION NUMBER | REVISION DATE | DESCRIPTION | PAGES CHANGED |
|---|---|---|---|
| 0 | 1/05 | Initial release. | — |
| 1 | 2/05 | Changed Digital Temp Sensor Output from ±2°C to ±3°C. | 1, 3 |
| | | Updated *Typical Operating Circuit*. | 1 |
| | | Changed $T_A$ = -40°C to +85°C to $T_A$ = $T_{MIN}$ to $T_{MAX}$. | 2, 3, 4 |
| | | Updated *Block Diagram*. | 8 |
| 2 | 6/05 | Added “UL Recognized” to Features; added lead-free packages and removed S from top mark info in *Ordering Information* table; added ground connections to the N.C. pin in the Typical Operating Circuit. | 1 |
| | | Added “noncondensing” to operating temperature range; changed $V_{PF}$ MIN from 2.35V to 2.45V. | 2 |
| | | Added aging offset specification. | 3 |
| | | Relabeled TOC4. | 7 |
| | | Added arrow showing input on X1 in the *Block Diagram*. | 8 |
| | | Updated pin descriptions for $V_{CC}$ and $V_{BAT}$. | 9 |
| | | Added the I²C Interface section. | 10 |
| | | *Figure 1:* Added sign bit to aging and temperature registers; added MSB and LSB. | 11 |
| | | Corrected title for rate select bits frequency table. | 13 |
| | | Added note that frequency stability over temperature spec is with aging offset register = 00h; changed bit 7 from Data to Sign (Crystal Aging Offset Register). | 14 |
| | | Changed bit 7 from Data to Sign (Temperature Register); correct pin definitions in *I²C Serial Data Bus* section. | 15 |
| | | Modified the *Handing*, *PC Board Layout*, and *Assembly* section to refer to J-STD-020 for reflow profiles for lead-free and leaded packages. | 17 |
| 3 | 11/05 | Changed lead-free packages to RoHS-compliant packages. | 1 |
| 4 | 10/06 | Changed $\overline{RST}$ and UL bullets in *Features*. | 1 |
| | | Changed EC condition “$V_{CC}$ > $V_{BAT}$” to “$V_{CC}$ = Active Supply (see Table 1).” | 2, 3 |
| | | Modified Note 12 to correct $t_{REC}$ operation. | 6 |
| | | Added various conditions text to TOCs 1, 2, and 3. | 7 |
| | | Added text to pin descriptions for 32kHz, $V_{CC}$, and $\overline{RST}$. | 9 |
| | | Table 1: Changed column heading “Powered By” to “Active Supply”; changed “applied” to “exceeds $V_{PF}$” in the *Power Control* section. | 10 |
| | | Indicated BBSQW applies to both SQW and interrupts; simplified temp convert description (bit 5); added “output” to $\overline{INT}$/SQW (bit 2). | 13 |
| | | Changed the *Crystal Aging* section to the *Aging Offset* section; changed “this bit indicates” to “this bit controls” for the enable 32kHz output bit. | 14 |
| 5 | 4/08 | Added Warning note to EC table notes; updated Note 12. | 6 |
| | | Updated the *Typical Operating Characteristics* graphs. | 7 |
| | | In the *Power Control* section, added information about the POR state of the time and date registers; in the *Real-Time Clock* section, added to the description of the RST function. | 10 |
| | | In Figure 1, corrected the months date range for 04h from 00–31 to 01–31. | 11 |

## Revision History (continued)

| REVISION NUMBER | REVISION DATE | DESCRIPTION | PAGES CHANGED |
|---|---|---|---|
| 6 | 10/08 | Updated the *Typical Operating Circuit*. | 1 |
| | | Removed the $V_{PU}$ parameter from the *Recommended DC Operating Conditions* table and added verbiage about the pullup to the *Pin Description* table for $\overline{INT}$/SQW, SDA, and SCL. | 2, 9 |
| | | Added the Delta Time and Frequency vs. Temperature graph in the *Typical Operating Characteristics* section. | 7 |
| | | Updated the *Block Diagram*. | 8 |
| | | Added the *$V_{BAT}$ Operation* section, improved some sections of text for the *32kHz TCXO* and *Pushbutton Reset Function* sections. | 10 |
| | | Added the register bit POR values to the register tables. | 13, 14, 15 |
| | | Updated the *Aging Offset* and *Temperature Registers (11h–12h)* sections. | 14, 15 |
| | | Updated the I2C timing diagrams (Figures 3, 4, and 5). | 16, 17 |
| 7 | 3/10 | Removed the “S” from the top mark in the *Ordering Information* table and the *Pin Configuration* to match the packaging engineering marking specification. | 1, 18 |
| 8 | 7/10 | Updated the *Typical Operating Circuit*; removed the “Top Mark” column from the *Ordering Information*; in the *Absolute Maximum Ratings* section, added the theta-JA and theta-JC thermal resistances and Note 1, and changed the soldering temperature to +260°C (lead(Pb)-free) and +240°C (leaded); updated the functional description of the $V_{BAT}$ pin in the *Pin Description*; changed the timekeeping registers 02h, 09h, and 0Ch to “20 Hour” in Bit 5 of Figure 1; updated the BBSQW bit description in the *Control Register (0Eh)* section; added the land pattern no. to the *Package Information* table. | 1, 2, 3, 4, 6, 9, 11, 12, 13, 18 |
| 9 | 1/13 | Updated *Absolute Maximum Ratings*, and last paragraph in *Power Control* section | 2, 10 |
| 10 | 3/15 | Revised *Benefits and Features* section. | 1 |

# Anexo 4

# Manual de usuario para la librería RTC_lib

# MANUAL DE USUARIO DE LA LIBRERÍA RTC_LIB

Pedro José Vera Gallego
UNIVERSIDAD DE VALLADOLID

# 1- Resumen de la funcionalidad de la librería

La librería RTC_lib, ha sido diseñada para el manejo de los dispositivos RTC de la marca Dallas, en concreto los dispositivos que utilizan el bus I2C, DS1307-DS3231.

Con esta librería es posible:

- Establecer la fecha y hora deseada.
- Leer la fecha y hora actual del RTC.
- Variar la fecha o la hora (un dato en concreto y en una unidad).
- Establecer distintos tipos de alarma.

# 2- Lista de constantes definidas

- ON → 1
- OFF → 0
- PM → 1
- AM → 0
- FORMATO_LARGO → 1
- FORMATO_CORTO → 2
- LUNES → 1
- MARTES → 2
- MIERCOLES → 3
- JUEVES → 4
- VIERNES → 5
- SABADO → 6
- DOMINGO → 7

## 3- Lista de funciones

- RTC
- leerHora ()
- Modo12h (AM_PM)
- escribeHora (hora, min, seg)
- escribeFecha (dia, mes, anno)
- escribeDiaSem (diaSem)
- variaHora (varia)
- variaMin (varia)
- variaAnno (varia)
- variaMes (varia)
- variaDia (varia)
- variaDiaSem (varia)
- verHora (formato)
- verFecha (formato, orden)
- verDia (formato)
- verMes (formato)
- verTemperatura ()
- alarmasON (tipo)
- escribeAlarma (dia, hora, min, seg)
- limpiaAlarma ()

## 4- Detalles de cada función

- RTC

  Con esta sentencia se define la clase. Es necesario declararla al inicio del programa para el uso del resto de funciones.

  *p. ej:* RTC rtc;

- leerHora

  Función que de tipo Reloj, estructura que contiene los valores de día, mes, año, dia de la semana, hora, hora formato 12h, minutos, segundos y cambio horario.

  No recibe ningún argumento y devuelve una estructura tipo Reloj.

  p. *ej:* Reloj x = leerHora ();

- Modo12h (AM_PM)

  Función para elegir el tipo de modo horario a utilizar, de 12 horas o de 24 horas. No devuelve ningún valor.

  Se recibe como argumento AM_PM, que es una variable booleana, la cual será 1 si se quiere el modo 12 horas y 0 si se quiere el modo 24 horas.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* Modo12h (OFF);

- escribeHora (hora, min, seg)

  Con esta función se inicializan, o simplemente se modifican, los registros de hora, minutos y segundos del RTC. No devuelve ningún valor.

  Los argumentos recibidos son la hora en primer lugar, después minutos y finalmente segundos.

  Es posible no escribir alguno de los valores, si el inferior no se ha escrito, siendo por defecto 0. Si un valor no se escribe y el siguiente valor si ha escrito, se producirá error de compilación.

  *p. ej:* escribeHora (10, 25); //Se establecerían las 10:25:00

- escribeFecha (dia, mes, año)

  Con esta función se inicializan, o simplemente se modifican, los registros de días, meses y años del RTC. No devuelve ningún valor.

  Los argumentos recibidos son el día en primer lugar, después el mes y finalmente el año.

  Es posible no escribir alguno de los valores, si el siguiente no se ha escrito, siendo por defecto 0. Si un valor no se escribe y el siguiente valor si ha escrito, se producirá error de compilación.

  *p. ej:* escribeFecha (6, 7, 2017); //Se establecería 6/07/2017

- escribeDiaSem (diaSem)

  Con esta función se modifica el registro de los días de la semana del RTC. No devuelve ningún valor.

  Se recibe como argumento el día de la semana.

  Se pueden emplear las constantes de los días de la semana.

  *p. ej:* escribeDiaSem (LUNES);

- variaHora (varia)

  Con esta función se modifica el valor del registro de la hora en una unidad.

  Se recibe como argumento una variable booleana, que será 1 si se quiere aumentar el valor o 0 si se quiere decrementar.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* variaHora (ON);

- variaMin (varia)

  Con esta función se modifica el valor del registro de los minutos en una unidad.

  Se recibe como argumento una variable booleana, que será 1 si se quiere aumentar el valor o 0 si se quiere decrementar.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* variaMin (ON);

- variaAnno (varia)

  Con esta función se modifica el valor del registro de los años en una unidad.

  Se recibe como argumento una variable booleana, que será 1 si se quiere aumentar el valor o 0 si se quiere decrementar.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* variaAnno (ON);

- variaMes (varia)

  Con esta función se modifica el valor del registro de mes en una unidad.

  Se recibe como argumento una variable booleana, que será 1 si se quiere aumentar el valor o 0 si se quiere decrementar.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* variaMes (ON);

- variaDia (varia)

  Con esta función se modifica el valor del registro de los días en una unidad.

  Se recibe como argumento una variable booleana, que será 1 si se quiere aumentar el valor o 0 si se quiere decrementar.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* variaDia (ON);

- variaDiaSem (varia)

  Con esta función se modifica el valor del registro de los días de la semana en una unidad.

  Se recibe como argumento una variable booleana, que será 1 si se quiere aumentar el valor o 0 si se quiere decrementar.

  Se pueden emplear las constantes ON y OFF.

  *p. ej:* variaDiaSem (ON);

- verHora (formato)

  Función que convierte los valores de los registros de horas, minutos y segundos en una cadena de caracteres, para ser entendible por el usuario. Se devuelve un String.

  Como argumento se recibe un entero para seleccionar el formato deseado, que será 1 si se quiere un formato largo, en que se muestran los segundos o 0 si se quiere un formato corto, sin los segundos. Si no se añade, por defecto el formato es largo.

  Se pueden emplear las constantes FORMATO_LARGO y FORMATO_CORTO.

  *p. ej:* verHora (FORMATO_LARGO); //mostraría 00:00

- verFecha (formato, orden)

  Función que convierte los valores de los registros de día, mes y año en una cadena de caracteres, para ser entendible por el usuario. Se devuelve un String.

  Como argumento se recibe un entero para seleccionar el formato deseado, que será 1 si se quiere un formato largo, en que se muestran los segundos o 0 si se quiere un formato corto, sin los segundos.

  Se pueden emplear las constantes FORMATO_LARGO y FORMATO_CORTO.

  El otro parámetro que se pasa a la función es el orden, que selecciona el modo en que se quiere la cadena con los enteros:

  - 1 → Día/Mes/Año
  - 2 → Año/Mes/Día
  - 3 → Mes/Día/Año

  *p. ej:* verFecha (FORMATO_LARGO, 1); //mostraría 6/07/2017

- verDia (formato)

  Función que convierte el valor del registro día de la semana en una cadena de caracteres, para ser entendible por el usuario. Se devuelve un String.

  Como argumento se recibe un entero para seleccionar el formato deseado, que será 1 si se quiere un formato largo, en que se muestran los segundos o 0 si se quiere un formato corto, sin los segundos.

  Se pueden emplear las constantes FORMATO_LARGO y FORMATO_CORTO.

  *p. ej:* verDia (FORMATO_CORTO); //mostraría Lun

- verMes (formato)

  Función que convierte el valor del registro mes en una cadena de caracteres, para ser entendible por el usuario. Se devuelve un String.

  Como argumento se recibe un entero para seleccionar el formato deseado, que será 1 si se quiere un formato largo, en que se muestran los segundos o 0 si se quiere un formato corto, sin los segundos.

  Se pueden emplear las constantes FORMATO_LARGO y FORMATO_CORTO.

  *p. ej:* verMes (FORMATO_LARGO); //mostraría Julio

- verTemperatura ()

  Función que lee los valores almacenado en los registros de temperatura y devuelve un float con el valor de la temperatura en grados Celsius.

  *p. ej:* *float* temp = verTemperatura ();

- alarmasON (tipo)

  Con esta función se activan las alarmas y se selecciona el tipo de alarma que se quiere activar. No devuelve ningún valor.

  Como argumento se pasa un entero entre el 1 y el 8, que indica el tipo de alarma a usar. Las 4 primeras corresponden a la alarma 1 y las siguientes 4 a la alarma 2. Si no se añade será 8. Los tipos de alarma son los siguientes:

  - 1 → Alarma cada segundo.
  - 2 → Coincidencia de segundos con la alarma establecida.
  - 3 → Coincidencia de minutos y segundos.
  - 4 → Coincidencia de hora, minutos y segundos.
  - 5 → Alarma cada minuto.
  - 6 → Coincidencia de hora y minutos.
  - 7 → Coincidencia de día de la semana, hora y minutos.
  - 8 → Coincidencia de día del mes, hora y minutos.

  *p. ej:* alarmasON (4); //Alarma al coincidir hora/min/seg

- escibeAlarma (día, hora, min, seg)

  Con esta función se establece el instante en que se quiere forzar la alarma. No devuelve ningún valor.

  Esta función debe ser precedida por la función alarmasON.

  Como argumentos se reciben cuatro enteros, el día o día de la semana, dependiendo de la alarma establecida, la hora, los minutos y los segundos.

  En caso de añadir un campo que el tipo de alarma no necesita, no pasa nada, simplemente no se tendrá en cuenta.

  Es posible no incluir uno de los argumentos, si el siguiente no se ha añadido. Cuando no se añada un parámetro, será por defecto 0 para las horas, minutos y segundos, y 1 para el día.

  *p. ej:* escibeAlarma (10,15,10); //Alarma tipo 8, el 10 a las 15:10

- limpiaAlarma ()

  Función necesaria para limpiar los flags de las alarmas.

  Cuando se activa una alarma, cambia el valor del registro de estado. Esta función reestablece el valor inicial de dicho registro.

  Es necesario emplear esta función una vez se haya producido la interrupción deseada por la alarma establecida.

  No recibe ni devuelve ningún valor.

  *p. ej:* limpiaAlarma ();

# 5- Precauciones para el correcto funcionamiento

- Es importante inicializar la clase con la RTC para que el compilador no detecte errores.
- Antes de realizar funciones que requieran un control de tiempo, asegurarse de inicializar la hora al valor deseado mediante escibeHora y escibeFecha.
- Al almacenar valores de los registros, asegurarse que la lectura se asigna a una variable tipo Reloj.
- Emplear la función alarmasON con el tipo correcto de alarma que se desee establecer y a continuación establecerla con escibeAlarma.
- Asegurarse de usar limpiaAlarma tras producirse la interrupción.
- No emplear la función delay en el código fuente de nuestro proyecto, ya que las interrupciones no funcionarán correctamente.
- Asegurarse de que las variables que se vayan a modificar dentro de la interrupción a la que llama la alarma se declaran como volatile.