Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería en Electrónica Industrial y Automática

Programación y control de un reloj de tiempo real con Raspberry Pi

Autor:

Martín Montero, Alberto

Tutor:

Plaza Pérez, Francisco José

Departamento de Tecnología Electrónica

Valladolid, marzo de 2018.

AGRADECIMIENTOS

A mi familia, en especial a mis padres, por su apoyo incondicional, tanto económico como emocional. Sin vosotros no hubiera sido posible llegar hasta aquí.

A todas las personas que han formado parte de esta etapa.

A cada uno de los profesores que han conseguido desarrollar mi afición por esta carrera.

RESUMEN

En el presente trabajo se ha realizado la programación en Python de un módulo con las funciones necesarias para establecer la comunicación entre el módulo RTC DS3231 y Raspberry Pi para el envío y recepción de datos. Con ello se consigue un control de cada una de las funcionalidades de las que dispone dicho módulo. Así mismo, se proponen algunas aplicaciones prácticas.

PALABRAS CLAVE:

Raspberry Pi, Comunicación I^2C, Reloj de Tiempo Real, DS3231, Python.

ABSTRACT

In this work it has been developed a Python module which contains the main functions to communicate the DS3231 RTC module and Raspberry Pi for data transfer. This makes possible the control of all of the available functionalities of the module. Also, some practical applications are proposed.

KEYWORDS:

Raspberry Pi, I^2C Communication, Real Time Clock, DS3231, Python.

ÍNDICE DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

ÍNIDICE DE TABLAS

1. INTRODUCCIÓN

1.1. Antecedentes y justificación del proyecto

En la actualidad, son muchos los sistemas que requieren información de fecha y hora durante su tiempo de ejecución. El ejemplo más típico es el de un ordenador.

Dicha información se puede obtener mediante la conexión a un servidor web, pero para ello es necesario disponer, obviamente, de conexión a internet. En el caso en el que no sea posible disponer de dicha conexión a internet, el modo de obtener los datos de fecha y hora es a través de un reloj de tiempo real o RTC (siglas del término inglés *Real Time Clock*).

Un reloj de tiempo real no es más que un módulo que se compone, a grandes rasgos, de un cristal oscilador que genera una onda cuadrada y un chip encargado de, a partir de esta señal, mantener actualizados los registros que contienen la información de segundos, minutos, horas, día de la semana, día del mes, mes y año.

1.2. Objetivo del proyecto

El objetivo del presente trabajo es el desarrollo de un módulo en Python (equivalente a las librerías en C) que contenga las funciones necesarias para que el usuario pueda establecer la comunicación con un reloj de tiempo real (DS3231) y, con ello, permitir el intercambio de información, así como la programación de alarmas para determinados eventos activados por tiempo.

Este módulo aprovecha todas las funcionalidades que brinda el DS3231: modificar los registros para actualizar los datos de fecha y hora, leer dichos registros para conocer hora y fecha actuales y configurar alarmas para provocar eventualidades, entre otras.

Además, se va a realizar un pequeño estudio económico para establecer una aproximación a lo que supondría la implementación de lo desarrollado en este trabajo a los sistemas disponibles actualmente en el mercado.

1.3. Estructura de la memoria

A continuación, se va a realizar una breve descripción de cada uno de los puntos en los que se ha dividido esta memoria para entender un poco mejor el desarrollo de la misma.

INTRODUCCIÓN: se realiza una descripción del contexto en el que se va a desarrollar el trabajo, así como de los objetivos que se pretenden alcanzar con la elaboración del mismo.

ASPECTOS PREVIOS: marco teórico de cada una de las técnicas y elementos que intervienen en la elaboración del proyecto.

Comunicación I²C: descripción detallada de este protocolo de comunicación y de sus funciones básicas de escritura y lectura de datos.

Python: justificación del uso de este lenguaje de programación y descripción de funcionalidades necesarias para el desarrollo del proyecto.

Raspberry Pi: características generales del dispositivo y guía paso a paso para la configuración de los elementos necesarios para su uso (sistema operativo, comunicación I²C…).

Reloj de tiempo real: enumeración y descripción de las características del módulo seleccionado. Diagrama de conexionado para la comunicación con la Raspberry Pi.

DESARROLLO DEL TFG: descripción de los pasos seguidos partiendo de funciones básicas hasta llegar al módulo final que contiene las funciones necesarias para llevar a cabo el control y la programación del reloj de tiempo real. Durante ese desarrollo se van elaborando pruebas de funcionamiento para asegurar que las funciones finales y el comportamiento global del módulo es el deseado.

APLICACIONES PRÁCTICAS: se exponen dos aplicaciones simples para ayudar a comprender la finalidad del módulo desarrollado y la utilidad que podría tener su implementación en sistemas actuales.

ESTUDIO ECONÓMICO: realización de un presupuesto aproximado para estimar la viabilidad de incluir este dispositivo a los sistemas que se encuentran actualmente en el mercado.

CONCLUSIONES: enumeración de las ideas finales alcanzadas tras la elaboración del proyecto en relación a las técnicas empleadas en la elaboración del mismo y el enfoque futuro de este.

BIBLIOGAFRÍA: enumeración de las fuentes bibliográficas consultadas para el desarrollo del proyecto.

ANEJOS: documentación adicional necesaria. Incluye el código del módulo desarrollado, la información técnica del módulo usado (datasheet) y el código de las aplicaciones prácticas descritas en el apartado 4.

2. ASPECTOS PREVIOS

2.1. Comunicación I²C

La comunicación I²C (acrónimo del término inglés “Inter-Integrated Circuit”) es un protocolo de comunicación entre dispositivos conectados por un bus serie de datos. Este bus serie se compone únicamente de dos líneas: una de ellas de datos (SDA), a través de la cual se realiza el envío bidireccional de información, y la otra con una señal de reloj (SCL) que marcará la velocidad de transferencia de dicha información.

Las dos líneas, SDA y SCL, son de tipo drenador abierto asociado a un transistor de efecto de campo o FET (análogo al de colector abierto de los transistores bipolares). Esto quiere decir que los pulsos de información que se envían a través de la línea de datos son de estado bajo, permaneciendo la línea en estado alto como nivel lógico normal. Debido a esto se incorpora una resistencia pull-up a cada una de las líneas del bus I²C. Estas resistencias serán comunes para todo el bus y tendrán valores de entre 1,8 kΩ y 47 kΩ dependiendo de la velocidad.

Los dispositivos conectados al bus I²C pueden actuar como maestros o como esclavos. El protocolo soporta sistemas con varios maestros, aunque normalmente sólo hay un maestro en el bus, y es este el único que puede iniciar la comunicación, y será, además, el que marcará la velocidad de la señal de reloj a través de la línea SCL. Cuando hay varios maestros, el protocolo establece prioridades entre ellos: si un maestro tiene acceso al bus, el resto no pueden establecer comunicación, por lo que tiene la limitación de que no es posible la transferencia de información entre maestros. El resto de dispositivos actúan como esclavos y su única función es la de responder a las peticiones del maestro.

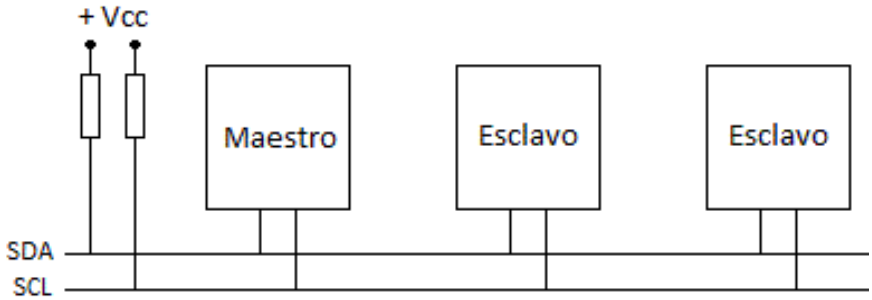

Ilustración 1. Ejemplo esquemático de un bus I2C

Existen varios modos de comunicación en función de la velocidad de transferencia. Es necesario fijar un modo compatible a todos los dispositivos ya que, por ejemplo, en el caso de un convertidor analógico-digital es necesaria una velocidad mínima para conseguir un correcto funcionamiento. En la Tabla 1 se

recogen los distintos modos de funcionamiento con los límites permisibles del reloj asociados a cada uno de ellos.

Modo	Velocidad de transmisión máxima
Standard Mode (Sm)	0,1 Mbit/s
Fast Mode (Fm)	0,4 Mbit/s
Fast Mode Plus (Fm+)	1,0 Mbit/s
High Speed Mode (HS-mode)	3,4 Mbit/s
Ultra-Fast Mode (UFm)	5,0 Mbit/s

Tabla 1. Modos de comunicación I2C

La unidad de información es un byte (8 bits) y un bit ACK. El bit ACK (Acknowledge) es un bit de confirmación que envía el esclavo al maestro en el noveno pulso de la señal de reloj poniendo un nivel lógico bajo en la línea de datos.

Cada uno de los dispositivos tiene asignada una dirección única. El bus I²C tiene un espacio de referencia de 7 bits de direcciones, con 16 direcciones reservadas, por lo que un único bus I²C puede contener hasta 112 dispositivos interconectados ($2^7 = 128,\ 128 - 16 = 112$ nodos). Cabe destacar que existe una limitación física debida a la capacidad de los buses, lo que restringe las líneas de comunicación a unos pocos metros.

El maestro establece la comunicación con un esclavo enviando en el byte la dirección propia de dicho dispositivo y este responde con una señal ACK para confirmar que ha recibido la petición del maestro. Normalmente las direcciones propias de cada dispositivo son de 7 bits, por lo que cuando se realiza el envío de una dirección hay un bit de información sobrante. Este bit es con el que el maestro informa sobre la acción a realizar, que puede ser lectura o escritura. Valdrá 1 cuando la acción sea la de lectura y 0 cuando se pretenda realizar una escritura de datos. Los siete bits de la dirección se envían en la parte más significativa del byte, y el bit de lectura/escritura ($R/\overline{W}$) será el bit menos significativo (LSB, Least Significant Bit).

Existen módulos que disponen de direcciones de 10 bits que son compatibles con el modo de direccionamiento de 7 bits usando 4 de las 16 direcciones reservadas siendo necesario, en esos casos, el envío de dos bytes para el direccionamiento. Ambos modos de direccionamiento se puede utilizar de manera simultánea. Con el direccionamiento de 10 bits se consigue aumentar el número máximo de dispositivos en un mismo bus hasta los 1136 nodos.

Los dos modos de direccionamiento se muestran en la Ilustración 2.

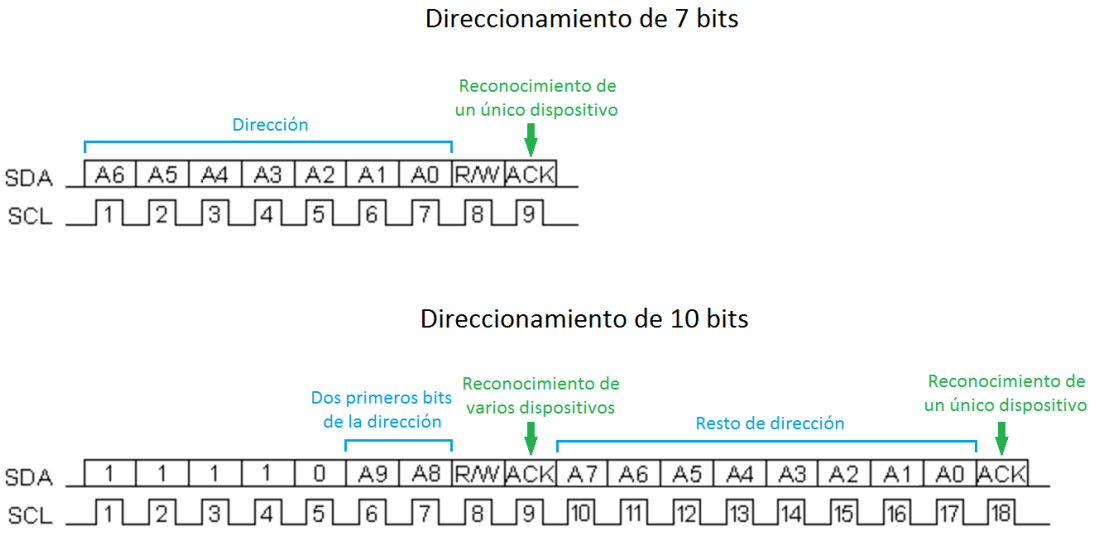

Ilustración 2. Modos de direccionamiento

Cuando ambas líneas se encuentran a nivel alto el bus está libre (Bus not busy). Para iniciar la comunicación cuando el bus está libre el maestro envía a la línea SDA el bit Start, que consiste en un cambio lógico de 1 a 0 en dicha de línea cuando la señal de reloj está a nivel alto. De manera análoga, finaliza la comunicación con un cambio de 0 a 1 cuando la señal de reloj este a nivel alto.

En la Ilustración 3 se muestran las secuencias de Start y Stop mencionadas anteriormente.

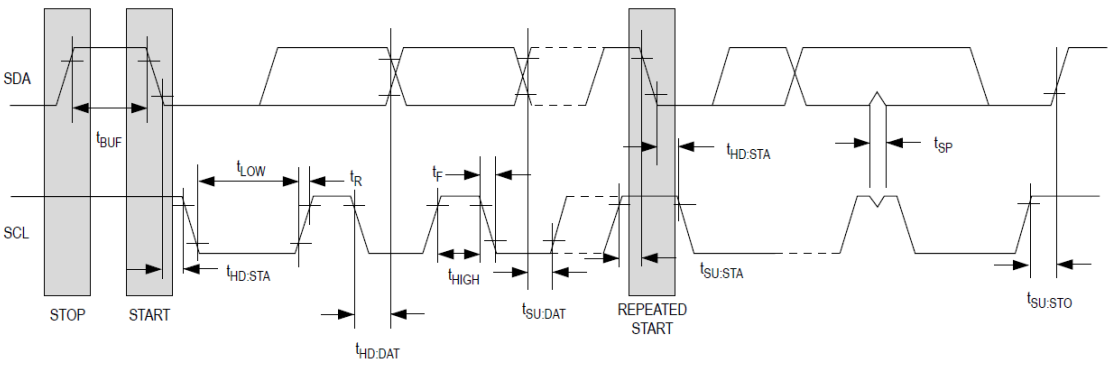

Ilustración 3. Secuencias de inicio y finalización de comunicación a través del bus

Una vez que el maestro envía la dirección del dispositivo con el que se desea comunicar y recibe la señal de reconocimiento de este, ya puede comenzar la transferencia de datos.

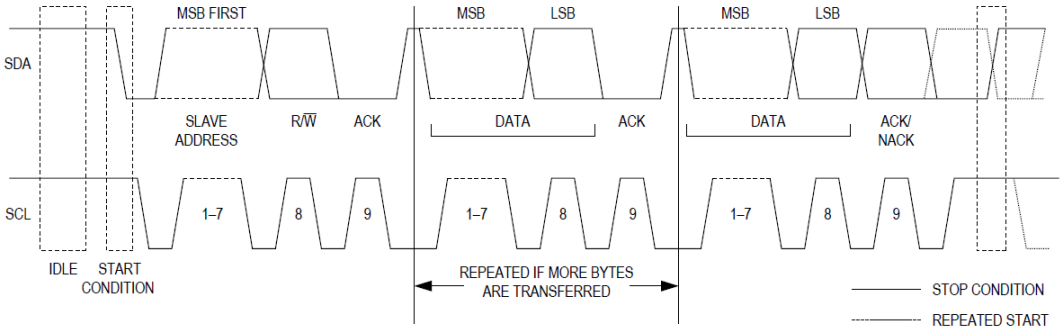

Ilustración 4. Secuencia de transferencia de datos a través del bus

A continuación se van a mostrar dos ejemplos de los dos modos comentados anteriormente: escritura y lectura de datos.

2.1.1. Escritura de datos

En el modo de escritura de datos el esclavo actúa como receptor y el maestro como emisor.

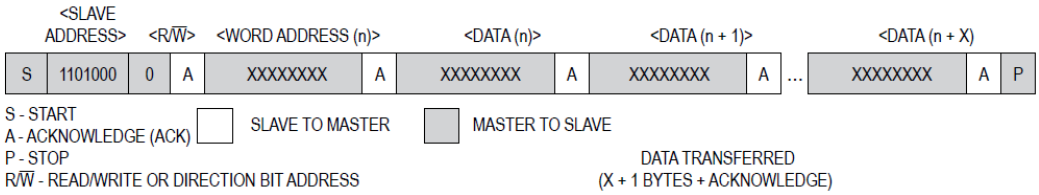

Ilustración 5. Escritura de datos a través del bus

En la Ilustración 5 se muestra de manera detallada la secuencia de escritura de datos en un esclavo por parte del dispositivo maestro a través del bus. La primera acción que realiza el maestro es la de iniciar la comunicación con el esclavo del que se quiere realizar la comunicación, por lo que, estando el bus libre, transmite el bit S (Start) y envía el primer byte. En este caso la dirección del esclavo es 1101000 y el bit $R/\overline{W}$ vale 0, ya que se pretende realizar una escritura de datos. Una vez recibida la confirmación del esclavo, el maestro envía la dirección del registro interno del esclavo en el que se desea realizar la escritura. Tras recibir la confirmación por parte del esclavo (bit A), el maestro comienza a mandar bytes de datos. El esclavo envía una señal de confirmación detrás de cada uno de ellos. Cuando finaliza la transmisión, el maestro cierra la comunicación con el bit P (Stop) dejando libre el bus de nuevo.

2.1.2. Lectura de datos

En el modo de lectura de datos el esclavo actúa como emisor y el maestro como receptor.

La secuencia de lectura se muestra de manera detallada en la Ilustración 6.

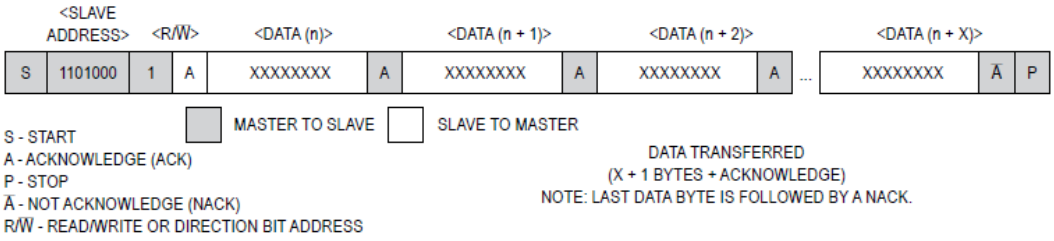

Ilustración 6. Lectura de datos a través del bus

Como se observa en la ilustración anterior, y al igual que en la escritura de datos, es el maestro el encargado de iniciar la comunicación con el esclavo deseado. Para ello, estando este libre, envía al bus el bit S (Start) seguido del primer byte que contiene la dirección del esclavo (1101000) y el bit $R/\overline{W}$ que, para este caso, tiene el valor lógico 1. Después de recibir la señal de confirmación por parte del esclavo, este comienza a enviar los byte de datos. El maestro envía señal de confirmación detrás de cada uno de ellos. Cuando finaliza la transmisión, el maestro envía un bit NACK ($\bar{A}$, Not Acknowledge) y cierra la comunicación con el bit P (Stop) dejando libre el bus de nuevo.

Existe otro método de lectura en el que se indica el registro del que se quiere obtener información. Este modo es algo más complejo ya que antes de realizar la lectura, el maestro tiene que realizar una escritura previa con el registro interno del esclavo del que quiere realizar la lectura.

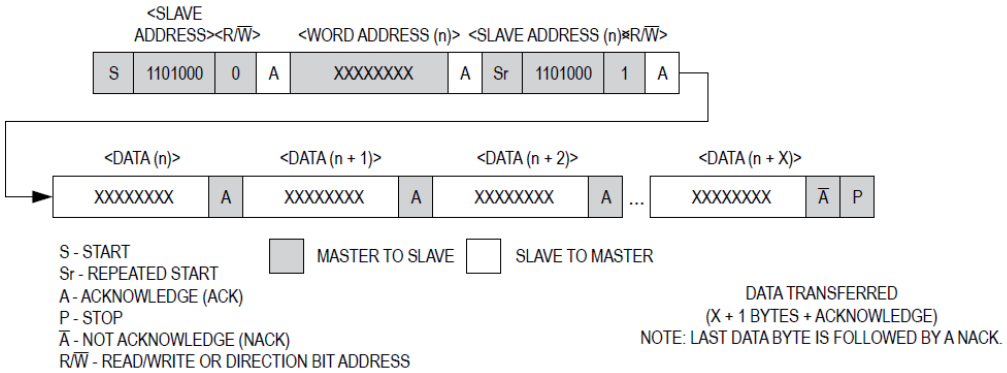

Ilustración 7. Otro modo de lectura de datos a través del bus

En la Ilustración 7 se muestra de manera detallada el otro modo de comunicación entre maestro y esclavo. Al igual que en el anterior, es el maestro el que inicia la comunicación con el bit S (Start) seguido del primer byte en el que envía la dirección del esclavo con el que se quiere realizar la comunicación y la acción a realizar. Este primer byte contiene la dirección del esclavo que, en este caso, es *1101000* y el bit $R/\overline{W}$ a 0 ya que, como se ha comentado antes, es necesario realizar una escritura previa por parte del maestro para indicar el registro interno del esclavo del que se quiere realizar la lectura. Una vez recibido el bit de confirmación por parte del esclavo, el maestro envía la dirección del registro interno y, una vez recibida la confirmación, envía el bit Sr que provoca que la comunicación se siga efectuando sin necesidad de cerrarla y volver a abrirla para efectuar la lectura. El maestro vuelve a enviar la dirección del esclavo del que quiere obtener información y seguido, ahora sí, el bit $R/\overline{W}$ a 1 para realizar la lectura. Cuando el maestro recibe la señal de confirmación por parte del esclavo pasa a ser receptor y el esclavo el emisor de la comunicación. El esclavo comienza a enviar byte de datos y el maestro envía bits de confirmación después de cada uno de ellos. Cuando ya se ha recibido toda la información el maestro el maestro envía un bit NACK ($\bar{A}$, Not Acknowledge) y vuelve a cerrar la comunicación con el bit P (Stop) dejando libre el bus.

En este modo de funcionamiento (lectura de datos) sigue siendo el maestro el que maneja la señal de reloj por lo que se plantea el siguiente problema: ¿Qué pasa si el esclavo no puede responder a tiempo con la información solicitada por el maestro? Esto podría suceder, por ejemplo, si el esclavo fuese un microcontrolador que está desempeñando varias funciones y cuando recibe la petición del maestro no es capaz de responder a tiempo.

El protocolo I²C proporciona una solución a este problema: el esclavo es capaz de mantener la línea SCL a nivel lógico bajo hasta que disponga de los datos solicitados. En ese instante coloca la información en el registro de transmisión y libera de nuevo la línea SCL. Esto se conoce como reloj de estiramiento.

La función del maestro en estos casos es la de emitir el primer pulso de reloj para la lectura colocándolo a nivel lógico alto. Después comprobará el estado de la línea y esperará hasta que vuelva a pasar a alto para continuar con la transmisión de datos.

NOTA: *la transferencia de datos en ambos casos se realiza con el bit más significativo (MSB) primero.*

2.2. Python

Python es un lenguaje de programación que apareció en 1991 de la mano de Guido van Rossum. Es un lenguaje de alto nivel, interpretado, orientado a objetos, con un enfoque muy simple y una sintaxis clara y limpia. Es multiplataforma, fuertemente tipado y usa un tipado dinámico.

2.2.1. ¿Por qué Python?

Como se acaba de mencionar, la principal característica de Python es su sencillez para obtener un código legible y de fácil comprensión debido a su sintaxis y, al ser un lenguaje interpretado, le dota de cierta flexibilidad frente a lenguajes compilados.

Python puede ser ejecutado en cualquier plataforma sin importar el sistema operativo que posea (a no ser que se empleen librerías específicas) ya que su intérprete está disponible en UNIX, Windows, Linux, Mac OS…

El ser un lenguaje orientado a objetos permite adaptar conceptos del mundo real a clases y objetos de programación para obtener interacciones entre estos.

Además, Python permite la programación imperativa y funcional, dado que es un lenguaje de programación multiparadigma.

Al poseer un tipado dinámico no es necesario declarar el tipo de variable sino que se determinará durante el tiempo de ejecución en función del dato que se le asigne a dicha variable, pudiendo cambiar de nuevo más adelante con una nueva asignación.

Por último, al ser un lenguaje fuertemente tipado no se puede tratar a una variable como si fuera de un tipo que no es, es necesario realizar un cambio de manera explícita. Por ejemplo, si una variable de tipo cadena contiene un número no podrá realizar una operación aritmética con otra de tipo número. Esta característica permite a Python no ser tan propenso a errores como otros lenguajes de programación.

Por todo esto es que Python es un lenguaje idóneo para desarrollo rápido de aplicaciones, pero sin perder efectividad en la programación.

Cabe destacar que Python no es recomendable para programación de bajo nivel o sistemas de rendimiento crítico, pero como no es el caso de lo que se va a tratar aquí, Python es una elección.

Existen dos versiones de Python: la 2 y la 3, que se diferencian únicamente en ligeros cambios en la sintaxis. Aquí se trabajará con la versión 3.

2.2.2. Instalar Python

La aplicación para trabajar con Python viene instalada de manera predeterminada con el sistema operativo Raspbian de la Raspberry Pi del cual se hablará más adelante, por lo que no será necesario realizar ninguna instalación.

2.2.3. Funciones I²C SMBus en Python

SMBus (del inglés System Management Bus) es un subconjunto del protocolo I²C que contiene las funciones básicas que se comentaron en el apartado 2.1. Por razones de compatibilidad es recomendable utilizar sólo estas funciones para controlar dispositivos I²C.

Las principales funciones se describen a continuación junto con toda su información relevante correspondiente (argumentos de entrada y datos que devuelve).

`long write_quick(int addr)`

Envía únicamente la dirección del dispositivo junto con el bit $R/\overline{W}$

- Argumentos de entrada: dirección del dispositivo (addr).
- Valores de retorno: variable tipo long

`long read_byte(int addr)`

Lee un byte del dispositivo especificado sin indicar el registro.

- Argumentos de entrada: dirección del dispositivo (addr).
- Valores de retorno: variable tipo long

`long read_byte_data(int addr, char cmd)`

Lee un byte del dispositivo especificado indicando también el registro interno del dispositivo del que se desea realizar la lectura.

- Argumentos de entrada: dirección del dispositivo (addr) y registro (cmd).
- Valores de retorno: variable tipo long

`long read_word_data(int addr, char cmd)`

Lee un “word” (2 bytes) del dispositivo especificado indicando el registro interno del dispositivo del que se quiere realizar la lectura.

- Argumentos de entrada: dirección del dispositivo (addr) y registro (cmd).
- Valores de retorno: variable tipo long

`long write_byte(int addr, char val)`

Envía un byte al dispositivo especificado sin indicar el registro.

- Argumentos de entrada: dirección del dispositivo (addr) y valor a enviar (val).
- Valores de retorno: variable tipo long

`long write_byte_data(int addr, char cmd, char val)`

Envía un byte al dispositivo especificado indicando el registro interno del dispositivo y el valor a enviar.

- Argumentos de entrada: dirección del dispositivo (addr), registro interno del dispositivo (cmd) y valor a enviar (val).
- Valores de retorno: variable tipo long

`long write_word_data(int addr, char cmd, int val)`

Envía un "word" (2 bytes) al dispositivo especificado indicando el registro interno del dispositivo y el valor a enviar.

- Argumentos de entrada: dirección del dispositivo (addr), registro interno del dispositivo (cmd) y valor a enviar (val).
- Valores de retorno: variable tipo long

Para este caso en particular únicamente se utilizarán las funciones *read_byte_data* y *write_byte_data* ya que, al enviar el registro interno del dispositivo, son más completas que *read_byte* y *write_byte*. Además, como todo los datos a enviar y los registros del módulo RTC son de un byte, las funciones *read_word_data* y *write_word_data* no van a ser útiles.

2.2.4. Definición de funciones

Las funciones son fragmentos de código con un nombre asociado que realiza una serie de tareas que pueden devolver un valor.

En Python las funciones se definen con la palabra clave "def" seguida del nombre de la función y los argumentos de entrada (si los hay) entre paréntesis y separados por comas. Al final de la línea de declaración de la función se añaden dos puntos (":").

Recordar que en Python no se hace uso de ningún tipo de elemento para englobar el contenido de una función (como podrían ser las llaves en C). Python reconoce que una serie de declaraciones o código pertenecen a una función mediante tabulaciones.

Un ejemplo de declaración de una función que imprima por pantalla *Hola mundo* sería el que se muestra a continuación:

```
def mi_funcion():
    print("Hola mundo")
```

Para realizar una llamada a la función anterior bastaría con, en cualquier línea del programa escribir:

```
mi_funcion()
```

NOTA: cuando las funciones se encuentran incluidas en una clase, la cual está asociada a un objeto (programación orientada a objetos), se las llama "métodos". La estructura sigue siendo la misma que las funciones, lo único que cambia es el paradigma de programación. Por esto mismo, de aquí en adelante se emplearán métodos en vez de funciones.

2.2.5. Módulos

Los programas en Python, para facilitar la lectura en caso de ser muy largos, pueden agruparse en módulos. Un módulo es un fragmento de código que engloban una serie de funciones, similares a las librerías en C.

Dentro de un programa, si se quisieran utilizar funciones que contiene un módulo en concreto, bastaría con importar dicho módulo al comienzo del programa.

Por ejemplo, para importar el módulo "mi_modulo" para utilizar las funciones que este contiene habría que incluir la siguiente línea al inicio del programa:

```
import mi_modulo
```

Suponiendo que la función "mi_funcion()" definida antes se encontrase dentro del módulo "mi_modulo" y se quisiera hacer uso de ella habría que usar (en el mismo programa en el que ya se ha importado dicho módulo:

```
mi_modulo.mi_funcion()
```

2.2.6. Importar módulo smbus

Para poder hacer uso de las funciones descritas en el apartado anterior será necesario importar el objeto *SMBus* que se encuentra en el módulo *smbus*. Para ello bastará con añadir al comienzo del módulo que se va a realizar la siguiente línea de código:

```
from smbus import SMBus
```

2.3. Raspberry Pi

La Raspberry Pi es, básicamente, un ordenador de bajo coste y bajo consumo (entre 2.5 y 3.5 watios) en una placa de reducido tamaño: 85.6 x 53.98 mm (aproximadamente las dimensiones de una tarjeta de crédito). Permite la utilización de diversos sistemas operativos como Raspbian, que es una versión Debian optimizada para Raspberry.

Actualmente existen varios modelos de Raspberry Pi: 1A+, 1 B+, 2B, Zero, Zero W y el 3B.

En este trabajo se usará el último modelo, el 3B, que se muestra en la Ilustración 8.

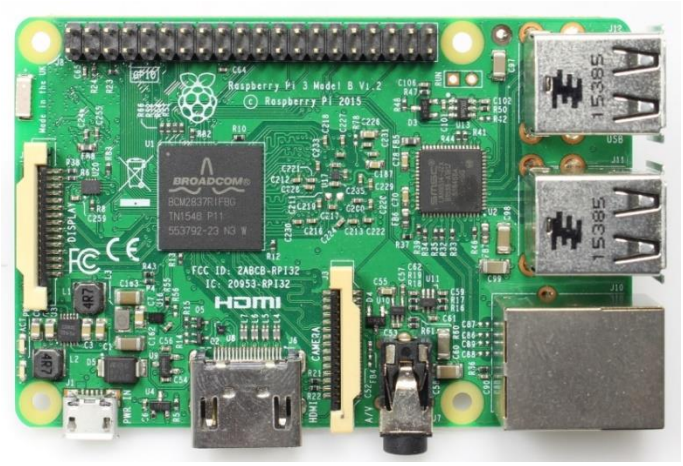

Ilustración 8. Raspberry Pi modelo 3B

2.3.1. Características técnicas

Como ya se ha mencionado, el modelo con el que se va a trabajar es el 3B, el cual cuenta con las siguientes características:

- Procesador ARMv8 de 64 bits con 4 núcleos a 1.2 GHz modelo BCM2837
- GPU Broadcom VideoCore IV a 400 MHz con soporte de OpenGL ES 2.0
- Memoria RAM de 1 Gb (compartido con la GPU)
- Conexión Wi-Fi 802.11n
- Bluetooth 4.1 de bajo consumo
- 4 puertos USB
- 40 pines GPIO
- Puerto HDMI
- Puerto Ethernet
- Puerto Jack 3.5mm para audio o salida de vídeo
- Almacenamiento interno por medio de tarjeta MicroSD
- Interfaz SDI para conectar un panel LCD

- Conector MIPI CSI para conectar un módulo de cámara

2.3.2. Instalación del sistema operativo

Por defecto, la Raspberry Pi viene sin un sistema operativo instalado. En este apartado se verá cómo realizar dicha instalación.

Para ello será necesario lo siguiente:

- Fuente de alimentación microUSB de 5V
- Tarjeta MicroSD de clase 10 (recomendable 8 Gb)
- Cable HDMI y monitor
- OPCIONAL: cable Ethernet RJ45

Lo primero que habrá que hacer será elegir el sistema operativo que se desee instalar en la Raspberry Pi.

Desde la página oficial de Raspberry se pueden descargar todos los sistemas operativos disponibles para Raspberry Pi (*https://www.raspberrypi.org/downloads/*). Bastará con descargar la imagen del sistema operativo seleccionado e instalarlo con ayuda de un asistente en la tarjeta MicroSD (previamente formateada) como podría ser Win32DiskImage.

Existe otra opción mucho más cómoda que sería descargar NOOBS desde esta misma página. NOOBS (acrónimo del término inglés “New Out Of Box Software”) es un asistente que nos permite instalar de manera más sencilla un sistema operativo en la Raspberry Pi. Únicamente habría que descargar el archivo ZIP desde la página oficial de Raspberry Pi, descomprimir el archivo y copiar todo el contenido en la tarjeta MicroSD. Una vez hecho esto bastará con insertar la tarjeta MicroSD en la Raspberry Pi y conectar la alimentación.

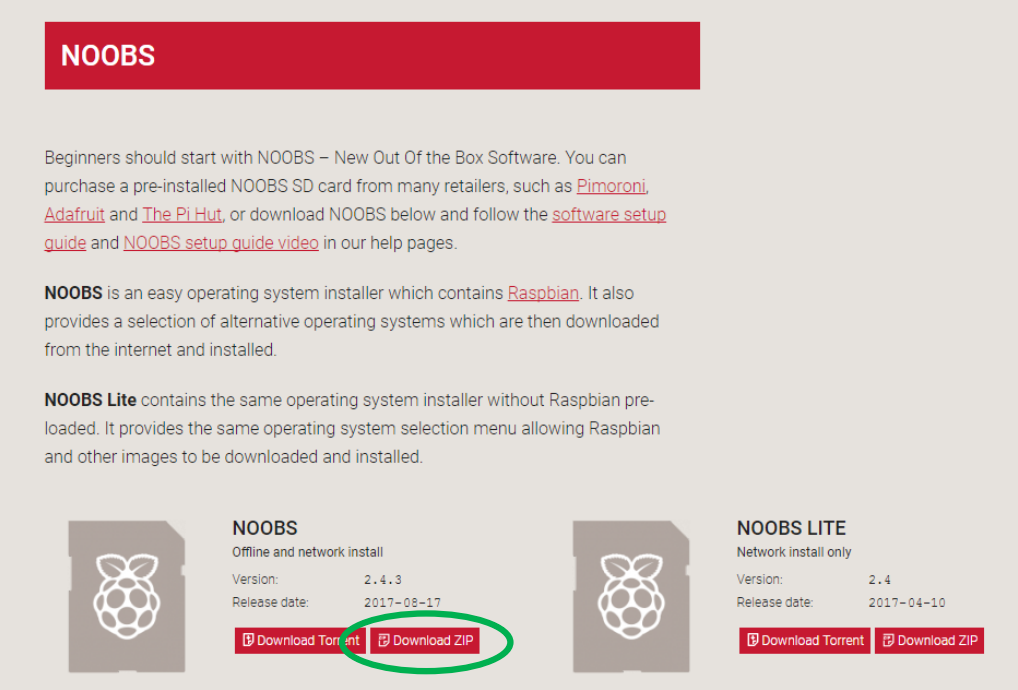

Ilustración 9. Descarga de NOOBS

Según se conecte la alimentación se encenderá la Raspberry Pi y con ello comenzará a ejecutarse el proceso de instalación. Seguidamente, aparecerá una ventana con una lista de sistemas operativos. Bastará con seleccionar el que se desee y comenzará la instalación.

Lo primero que pedirá será la configuración de conexión a internet para descargar paquetes y actualizaciones. En este punto es recomendable disponer de cable Ethernet para agilizar el proceso, aunque no es necesario. Cuando se descargue e instale todo, la Raspberry se reiniciará. En caso de que no lo haga, es recomendable hacerlo manualmente para que se aplique bien toda la configuración.

Una vez finalizada la instalación, la Raspberry Pi ya estará lista para su uso. En este punto se pueden realizar una serie de configuraciones a la Raspberry como, por ejemplo, cambiar la contraseña, cambiar idioma del sistema operativo, franja horaria, hacer overclock a la Raspberry, etc.

Para ello bastará escribir la siguiente línea en el terminal:

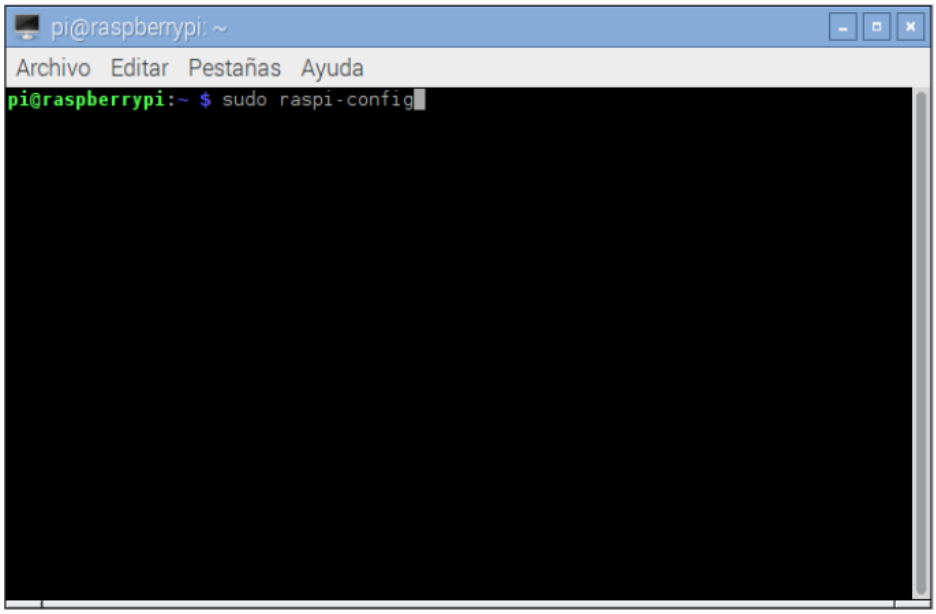

Ilustración 10. Acceso a la configuración desde terminal

Aparecerá la ventana de configuración que se muestra en la Ilustración 11.

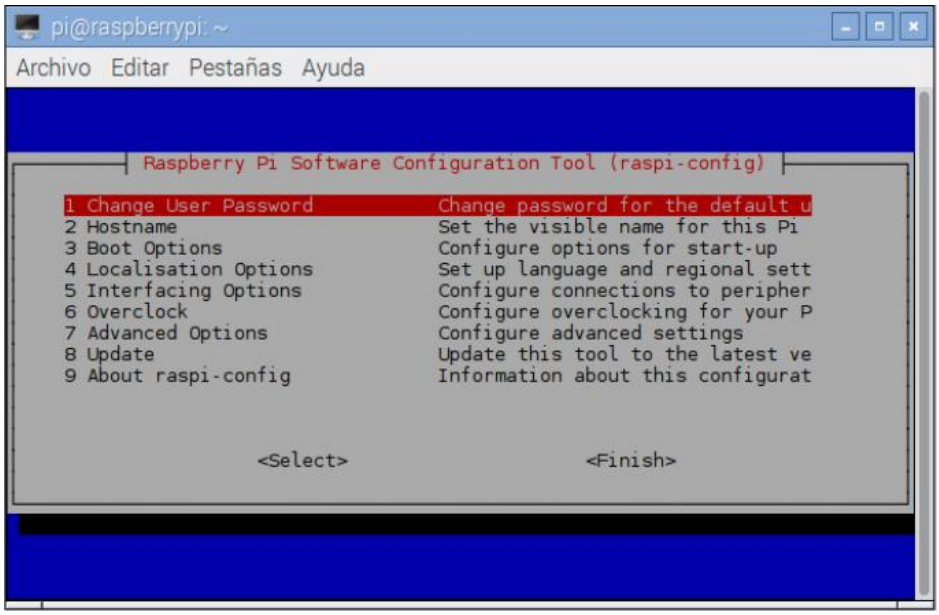

Ilustración 11. Menú de configuración

Una vez que está configurada la Raspberry Pi, habrá que hacer una serie de modificaciones para habilitar las comunicaciones de las que dispone (GPIO, SSH, VNC, I^2C...) ya que de manera predeterminada están todas “capadas”.

2.3.3. Trabajar con los GPIO

La Raspberry Pi 3 Modelo B dispone de 40 GPIO (*General Purpose Input/Output*) que, como su propio nombre indica, son pines que pueden usarse como entrada o salida para diversos usos. Con ellos se pueden elaborar circuitos como los que se pueden montar de manera análoga con Arduino.

Habilitar estos puertos será necesario, ya que mediante ellos se realiza la comunicación I2C de la que dispone Raspberry Pi.

A continuación se mostrará cómo habilitar dichos puertos con las librerías necesarias para poder controlarlos usando Python:

NOTA: Necesario trabajar con conexión a internet.

Se abre el terminal y se introducen las siguientes líneas:

```
sudo apt-get install python-dev
sudo apt-get install python-rpi.gpio
```

Una vez que se descarguen e instalen las librarías, se actualizan los repositorios y los programas, respectivamente:

```
sudo apt-get update && sudo apt-get upgrade
```

Por último, habrá que reiniciar la Raspberry Pi para que la configuración quede cargada:

```
sudo reboot
```

Con esto ya quedan habilitados los puertos GPIO.

En la Ilustración 12 se muestra la funcionalidad de cada uno de dichos puertos.

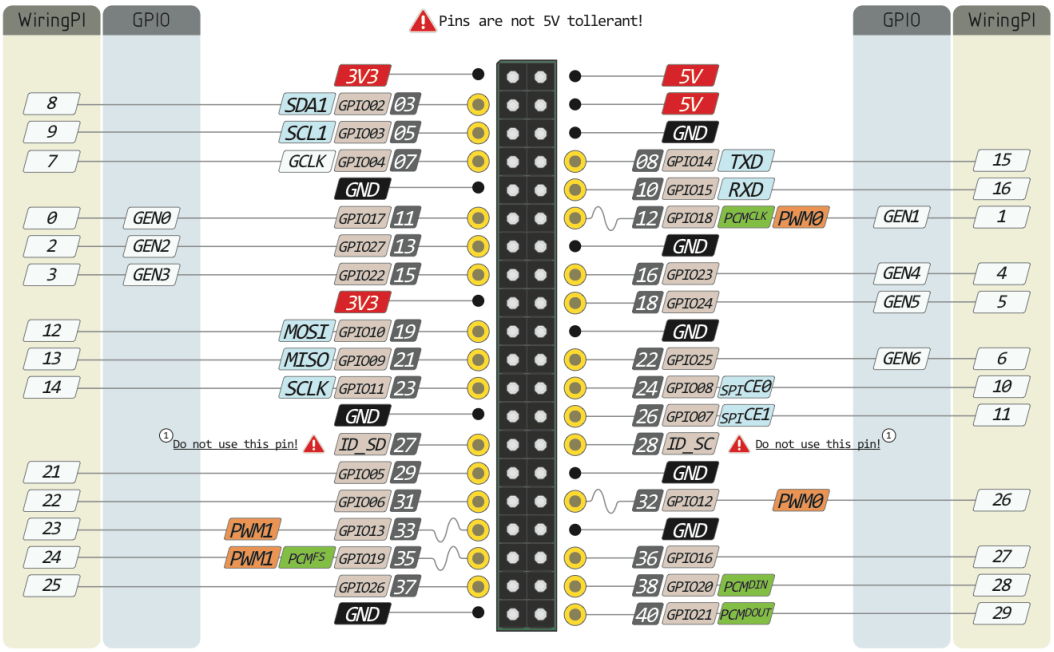

Ilustración 12. Información puertos GPIO

Antes de realizar alguna aplicación con los puertos GPIO hay que tener en cuenta que la Raspberry Pi trabaja con 3.3 V, por lo que si se introduce como entrada algún voltaje superior, se puede dañar la placa. Otro factor a tener en cuenta es que los pines de alimentación (3V3 y 5V) están limitados a 50 mA, por lo que si la aplicación requiere una corriente mayor, es necesario incluir una fuente de alimentación que suministre ese valor de corriente para evitar dañar la placa.

A continuación se muestra una lista de comandos básicos para configurar y trabajar con los GPIO usando Python:

Importar módulo

Para importar el módulo RPi.GPIO:

```
import RPi.GPIO as GPIO
```

Lo que se hace con el commando anterior es, además de importar el módulo RPi.GPIO, renombrarlo para que durante el resto del script de Python sólo sea necesario escribir *GPIO.nombre_funcion* para referirse a una función incluida en el módulo (como se vio en el apartado 2.2.5).

Numeración de los pines

Hay dos formas para nombrar los pines de la placa. Una de ellas es usar la numeración BOARD, que es la que se refiere a la posición que ocupa el pin en el *pin header*, desde el 1 hasta el 40. Usar este modo de numeración tiene una ventaja, y

es que un script se puede emplear en otra versión de Raspberry Pi sin necesidad de cambiar el código, ya que el hardware (posición de los pines) no varía.

La otra forma de numeración es la de los canales del chip *Broadcom*, la numeración BCM. Este modo de nombrar los pines es de más bajo nivel que la anterior y puede presentar problemas si un script se ejecuta en una placa de una versión diferente de placa para la que ha sido creado. (Ver Ilustración 12 para ambas numeraciones).

Ambos modos de numeración se definen como sigue:

```
GPIO.setmode(BOARD)
```

```
GPIO.setmode(BCM)
```

Habilitar avisos

Cuando se realiza la ejecución de un script que usa los GPIO, Raspberry detecta los cambios producidos en los puertos. La configuración predeterminada de un pin es como entrada, y si se configura como salida nos mostrará un aviso. Estos avisos se pueden eliminar con el siguiente comando:

```
GPIO.setwarnings(False)
```

Configuración de pines como entrada/salida

De manera similar a cómo se trabaja con Arduino, es necesario definir al comienzo del programa si un pin se configura como entrada o como salida.

Para configurar un pin como salida:

```
GPIO.setup(pin, GPIO.OUT)
```

Donde "pin" se refiere al pin con el que se va a trabajar. En función del modo de numeración de pines asignado será, por ejemplo, 17 para modo BCM o 11 para modo BOARD. (Ver Ilustración 12)

Si, por el contrario, se desea configurar un pin como entrada:

```
GPIO.setup(pin, GPIO.IN)
```

Además, Raspberry Pi ofrece la posibilidad de configurar resistencias pull-up/pull-down cuando se utilice un pin como entrada.

Para configurar entrada con resistencia pull-up:

```
GPIO.setup(pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Para configurar entrada con resistencia pull-down:

```
GPIO.setup(pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
```

Nuevamente, "pin" se refiere al pin que con el que se desea trabajar dependiendo del modo de numeración seleccionado previamente.

Leer entrada

Para leer el valor de entrada de un pin que ha sido configurado como tal:

```
GPIO.input(pin)
```

Donde "pin" se refiere al pin que con el que se desea trabajar dependiendo del modo de numeración seleccionado previamente.

Las entradas de Raspberry Pi con digitales, por lo que sólo habrá dos lecturas de entrada posibles: HIGH (True/1) o LOW (False/0).

Cambiar estado de una salida

Para cambiar el estado de un pin que ha sido configurado como salida:

```
GPIO.output(pin, estado)
```

Donde "pin" se refiere al pin que con el que se desea trabajar dependiendo del modo de numeración seleccionado previamente y "estado" el valor lógico que desea asignar a dicho pin. Puede ser *GPIO.HIGH* para un estado lógico alto (True/1) o *GPIO.LOW* para una estado lógico bajo (False/0).

"Limpiar" pines

Una vez que la ejecución del script vaya a finalizar, es buena práctica realizar una limpieza de los pines para evitar posibles daños a la placa. Esto quiere decir que los pines que se hayan usado en el script vuelven a la configuración predeterminada y se resetea el modo de numeración seleccionado. Para llevar a cabo dicha "limpieza":

```
GPIO.cleanup()
```

Obtener información de la placa y el módulo

Además, el módulo RPi.GPIO permite obtener información sobre la versión tanto de la placa Raspberry Pi como del módulo RPi.GPIO.

Para obtener la versión de la placa:

```
GPIO.RPI_REVISION
```

Para obtener la versión del módulo RPi.GPIO:

```
GPIO.VERSION
```

2.3.4. Habilitar comunicación I²C

Como se mencionó anteriormente, la configuración por defecto de la Raspberry Pi trae las comunicaciones deshabilitadas, por lo que será necesario configurar la comunicación I²C para poder trabajar con ella.

Lo primero de todo es eliminarlo de la lista negra, por lo que hay que acceder a dicho fichero de configuración y modificarlo. Para ello es necesario escribir la siguiente línea en el terminal:

```
sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

Una vez dentro del fichero habrá que comentar la última línea de este. El fichero finalmente debería quedar como se muestra en la Ilustración 13.

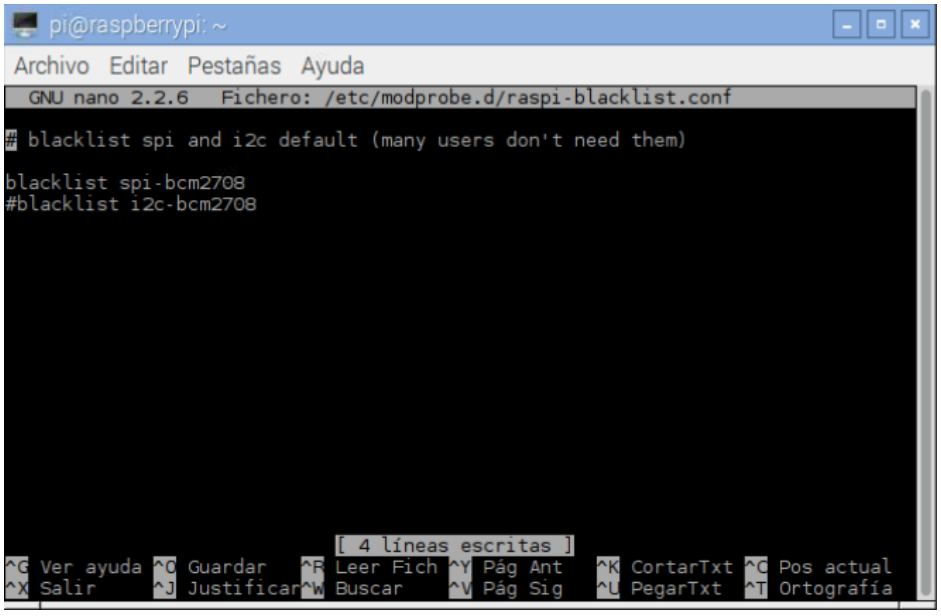

Ilustración 13. Modificación de la "blacklist"

Lo siguiente será añadir el módulo I²C al kernel. Para ello es necesario modificar otro fichero, al cual se accede del siguiente modo:

```
sudo nano /etc/modules
```

Una vez dentro de la edición del fichero, se añade la línea “i2c-dev” al final del mismo.

El fichero final quedaría como se muestra en la Ilustración 14.

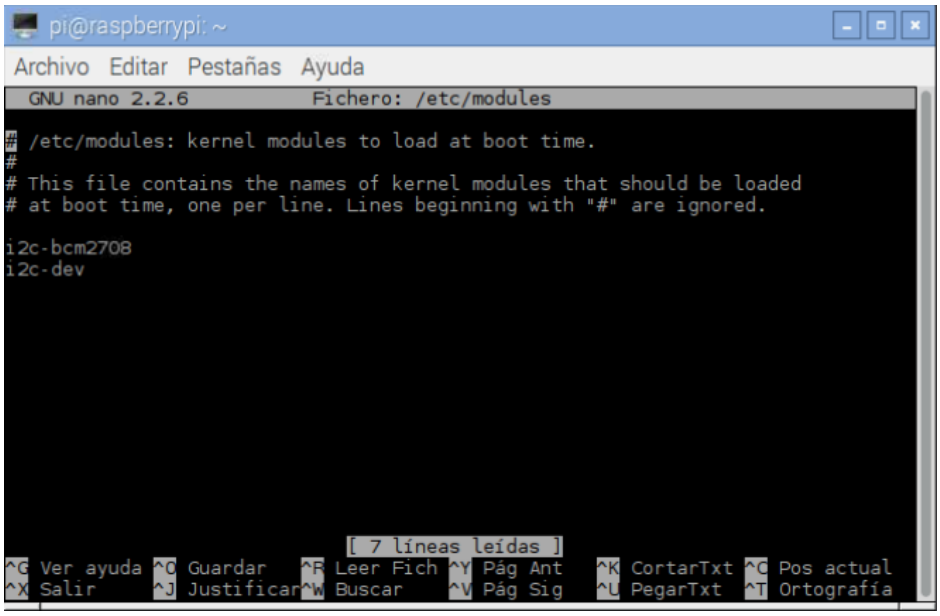

Ilustración 14. Modificación del fichero de módulos del kernel

Ahora es necesario descargar e instalar las herramientas y paquetes de I²C. Para ello será necesario disponer de conexión a internet.

El primer paquete se descarga con la siguiente línea en el terminal:

```
sudo apt-get install i2c-tools
```

En caso de que falle será necesario actualizar la Raspberry Pi y, después, volver a introducir la línea anterior. Para ello:

```
sudo apt-get update
```

El otro paquete que hay que instalar se consigue con el siguiente comando:

```
sudo apt-get install python-smbus
```

Ya están todos los paquetes instalados y lo único que queda es añadir al usuario de Raspberry Pi al grupo de acceso I²C para poder utilizar todas las funciones del protocolo. En este caso el usuario es “pi”. Esto se consigue con la siguiente línea de terminal:

```
sudo adduser pi i2c
```

Para acabar, es necesario reiniciar la Raspberry Pi para que todo la configuración quede cargada:

```
sudo reboot
```

Una vez que la Raspberry Pi se haya reiniciado ya se puede hacer uso de todas las herramientas I²C.

Para testear el software que se acaba de instalar se realiza una búsqueda de los dispositivos que están conectados al bus I²C con el siguiente comando:

```
i2cdetect -y 1
```

El comando anterior varía en función de la versión de Raspberry Pi que se esté utilizando. El mostrado es para los modelos B, mientras que para los modelos A sería:

```
i2cdetect -y 0
```

El resultado de búsqueda en caso de que hubiera ningún dispositivo conectado al bus se muestra en la Ilustración 15.

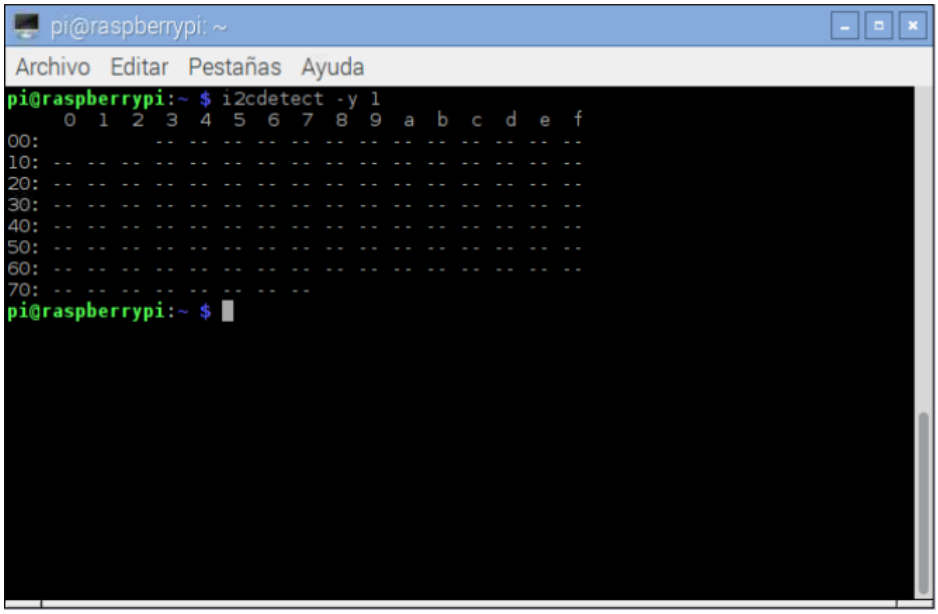

Ilustración 15. Búsqueda de dispositivos conectados al bus

En caso de haber algún dispositivo conectado al bus, al ejecuta el comando anterior se obtendrían las direcciones de cada uno de ellos. Un ejemplo de esto es el que se muestra en la Ilustración 16.

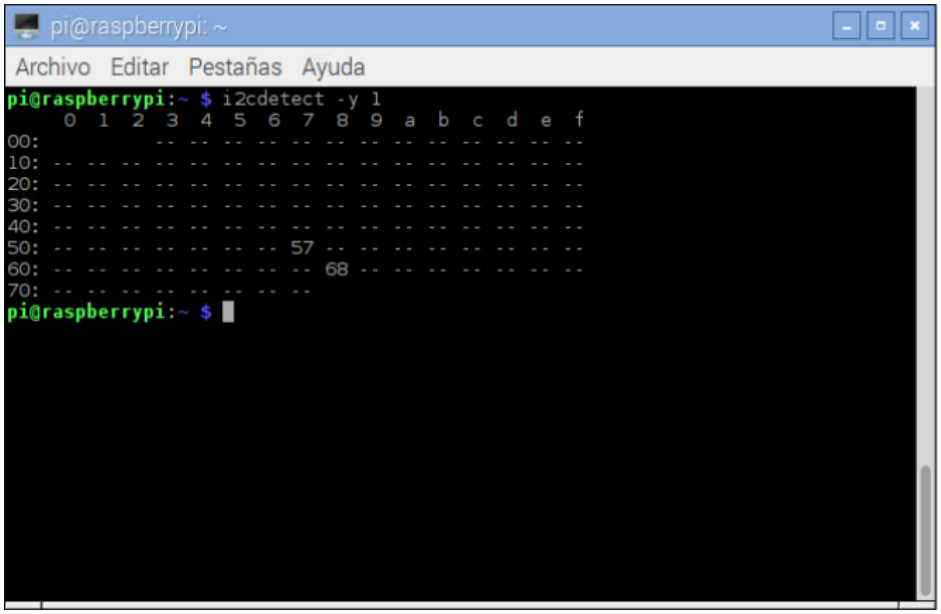

Ilustración 16. Escaneo de dispositivos conectados al bus

Como se observa en la Ilustración 16, hay dos dispositivos conectados al bus I²C cuyas direcciones son 0x57 (1010111) y 0x68 (1101000).

2.4. Reloj de tiempo real

Un reloj de tiempo real es un dispositivo electrónico encargado de mantener la hora y fecha actualizadas. Se compone de un cristal oscilador que genera una frecuencia de 32768 Hz y un chip que, a partir de dicha frecuencia, es capaz de mantener actualizados los registros internos en los que almacena los datos de fecha y hora como son los segundos, minutos, horas, día de la semana, día del mes, mes y año, entre otros.

El término "Reloj de Tiempo Real" se emplea para diferenciar este tipo de dispositivos de los relojes electrónicos habituales, los cuales sólo realizan un conteo de pulsos, sin unidades temporales.

La ventaja de usar estos chips reside en su reducido consumo de energía y el hecho de liberar de una carga de trabajo al procesador del sistema, sobre todo si se trata de uno que realiza operaciones críticas.

2.4.1. Selección del módulo

En el mercado existen diversos modelos de módulos RTC. Entre los principales fabricantes destacan: Phillips, Maxim Integrated y Texas Instruments, entre otros.

Dentro de los modelos de relojes de tiempo real más empleados para este tipo de proyectos en los que es necesaria la comunicación con microcontroladores como Arduino o Raspberry Pi, destacan el DS1307 y el DS3231, ambos de Maxim Integrated.

La principal diferencia entre ambos chips es la precisión: la del modelo DS3231 es mucho más superior que la del DS1307. Esto es debido a que el modelo DS3231 incorpora un sensor que permite realizar una medición de la temperatura para contrarrestar los efectos que producen las variaciones de esta, reduciendo significativamente los errores en las medidas temporales.

Así, mientras los datos del DS1307 sufren unos desfases temporales de uno o dos minutos diarios, el DS3231 está dotado de una precisión de 2 ppm, lo que se traduce en un desfase temporal de uno o dos segundos al mes, aproximadamente.

Además, el DS3231 trabaja con tensiones comprendidas entre 2.3 y 5.5 V, siendo 3.3 V el valor típico, mientras que el DS1307 trabaja con valores de tensión comprendidos entre 4.5 y 5.5 V. Si se recuerda lo mencionado en el apartado 2.3.3, los puertos GPIO trabajan con tensiones de 3.3 V, por lo que el modelo DS3231 es el idóneo para trabajar con la Raspberry Pi, ya que si se trabajase con el DS1307 sería necesario un convertidor de nivel lógico bidireccional para que la comunicación con la Raspberry Pi fuera posible sin llegar a dañar la placa.

Es por esto por lo que el modelo seleccionado para la realización de este proyecto es el DS3231, más concretamente la versión DS3231SN, que trabaja en un rango de temperaturas más amplio.

El módulo seleccionado con el que se va a trabajar, que lleva el chip DS3231 montado, es el ZS-042.

Ilustración 17. Chip DS3231

INCISO: Un convertidor de nivel lógico bidireccional es un dispositivo electrónico que se emplea para modificar (aumentar o reducir) el voltaje de señales electrónicas de baja tensión. Se emplea para poder establecer comunicación entre dispositivos que trabajen, por ejemplo, siguiendo el caso mencionado en el párrafo anterior, a 3.3 V y a 5 V.

Generalmente se suelen utilizar convertidores de 3.3 V a 5 V y viceversa, debido a los módulos y microcontroladores actuales, aunque existen otros modelos como los convertidores de 1.8 V a 2.8 V, y viceversa.

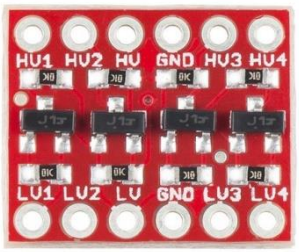

Ilustración 18. Convertidor de nivel lógico bidireccional

Como se observa en la Ilustración 18*, el convertidor tiene una parte HV (High Voltage) y otra LV (Low Voltage) con diversos canales en los que se establecerá la conversión. Por ejemplo, si se aplican 5 V en el canal uno del nivel lógico superior (HV1) se obtendrán 3.3 V en el canal uno del nivel lógico inferior (LV1), y viceversa. (Para el caso en el que el convertidor sea de 3.3-5 V)*

2.4.2. Características generales

Como se mencionó en el apartado anterior, el reloj de tiempo real seleccionado es el DS3231. El circuito típico que el fabricante recomienda para el uso de este chip se muestra en la Ilustración 19.

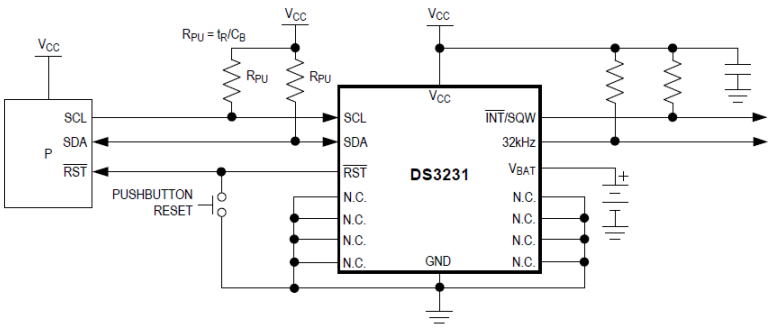

Ilustración 19. Circuito típico de trabajo con el DS3231

Como también se comentó en el apartado anterior, el módulo con el que se va a trabajar, el ZS-042, es el que aparece en la Ilustración 20.

Ilustración 20. Módulo ZS-042

A continuación se van a enumerar las principales características de este modelo en particular extraídas de su datasheet (Ver ANEXO):

- Trabaja para tensiones comprendidas entre los 2.3 V y los 5 V, siendo un valor típico el de 3.3 V. Valor óptimo para trabajar con la Raspberry Pi.
- Incorpora un sensor de medición de temperatura para compensación de errores debidos a cambios de esta. Dicho sensor almacena los valores de temperatura en los registros internos, por lo que será posible realizar lecturas de temperatura con el dispositivo, aunque no muy precisas (Precisión de ±3°C).
- Trabaja en un rango de temperatura amplio: desde -40°C hasta 85°C.
- Posee una alta precisión dentro del rango de temperaturas mencionado en el punto anterior, siendo de 2 ppm entre los 0°C y los 40 °C, y una precisión de 3 ppm entre los -40°C y los 85°C.
- El consumo de corriente es reducido: 200 µA cuando está activo y 110 µA cuando está en *standby*.

- Incorpora dos alarmas programables con diversos modos de funcionamiento.

  Para la alarma 1:
  - Alarma cada segundo.
  - Alarma a los segundos programados.
  - Alarma a los minutos y segundos programados.
  - Alarma a la hora, minutos y segundos programados.
  - Alarma al día del mes, hora, minutos y segundos programados.
  - Alarma al día de la semana, hora, minutos y segundos programados.

  Para la alarma 2:
  - Alarma cada minuto (cuando segundos valgan 00).
  - Alarma a los minutos programados.
  - Alarma a la hora y minutos programados.
  - Alarma al día del mes, hora y minutos programados.
  - Alarma al día de la semana, hora y minutos programados.

  Dispone de un pin denominado "SQW" (*Ilustración 20*) que manda un pulso de nivel lógico bajo con el que, mediante conexionado a la Raspberry Pi, es posible programar una interrupción cuando se produzca una alarma.
- Incorpora un pin nombrado como "32K" (*Ilustración 20*) para poder obtener la señal generada por el cristal oscilador.
- El pin "SQW" también permite la programación de una señal cuadrada de salida. Como no entra dentro de los objetivos de este trabajo esta funcionalidad no se utilizará.
- La interfaz I^2C soporta el modo de transferencia *Fast Mode* (Ver tabla 1), lo que permite una comunicación con una velocidad de transferencia de 400 kHz.
- El módulo incorpora una memoria EEPROM modelo 24C32 de 32Kb, lo que equivale a 4096 registros de 8 bits. En un principio no se hará uso de esta memoria.

NOTA: tanto el pin SQW como 32K requieren de una resistencia pull-up que puede ser añadida físicamente o configurando la entrada de la Raspberry Pi como tal.

A continuación se muestra un diagrama de bloques del DS3231:

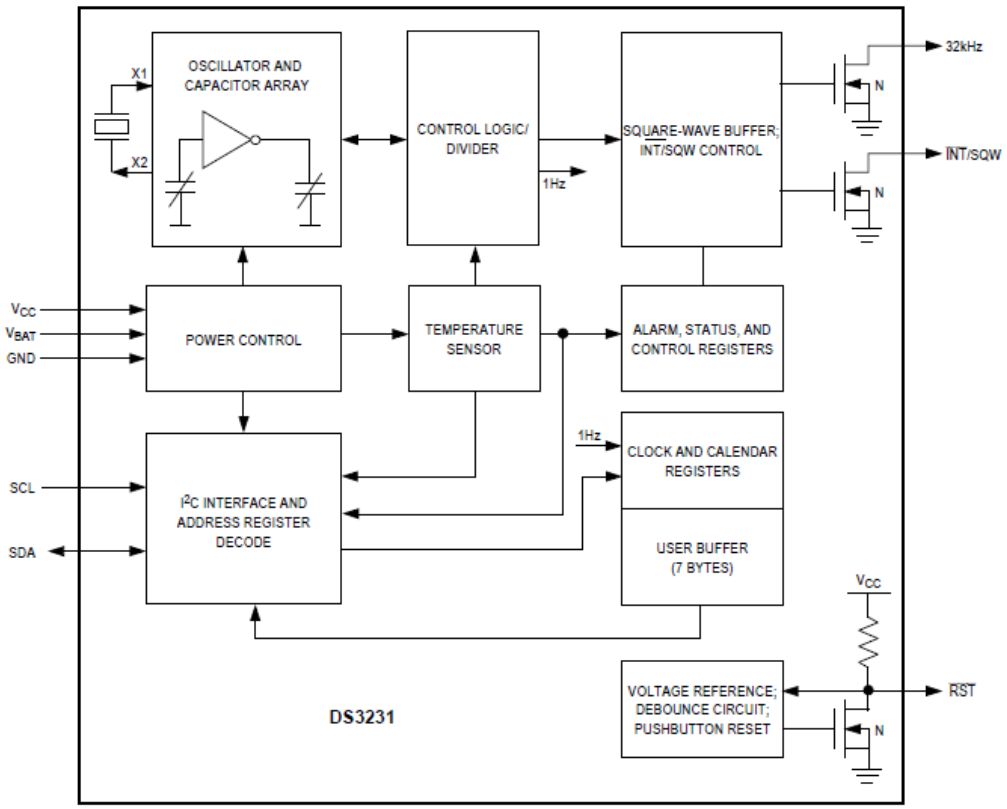

Ilustración 21. Diagrama de bloques del DS3231

Los bloques mostrados en la imagen anterior pueden ser agrupados en cuatro grupos funcionales: cristal oscilador con compensación de temperatura, control de alimentación, reinicio del dispositivo y reloj de tiempo real.

El cristal oscilador con compensación de temperatura engloba el sensor de temperatura, el oscilador y el control lógico.

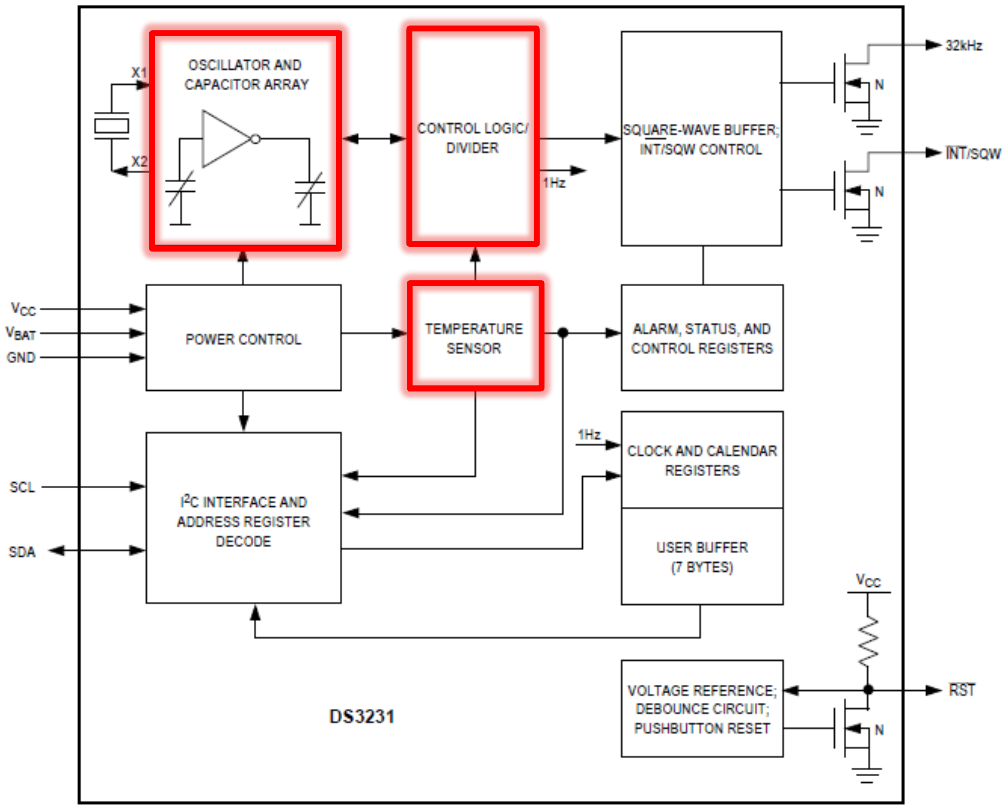

Ilustración 22. Grupo funcional del cristal oscilador

El chip realiza una lectura del sensor de temperatura y compara ese valor con unos datos tabulados para determinar la capacitancia requerida, añade la corrección de era en el registro de siglo y establece la selección de capacitancia en su registro. Todo esto ocurre únicamente cuando se produce un cambio de temperatura o cuando se completa una conversión de temperatura. Una conversión de temperatura se lleva a cabo cuando se alimenta al dispositivo con tensión externa (V_{CC}) o una vez cada 64 segundos.

La función de control de alimentación es proporcionada por una tensión de referencia compensada en función de la temperatura y un circuito comparador que controla el valor de V_{CC}.

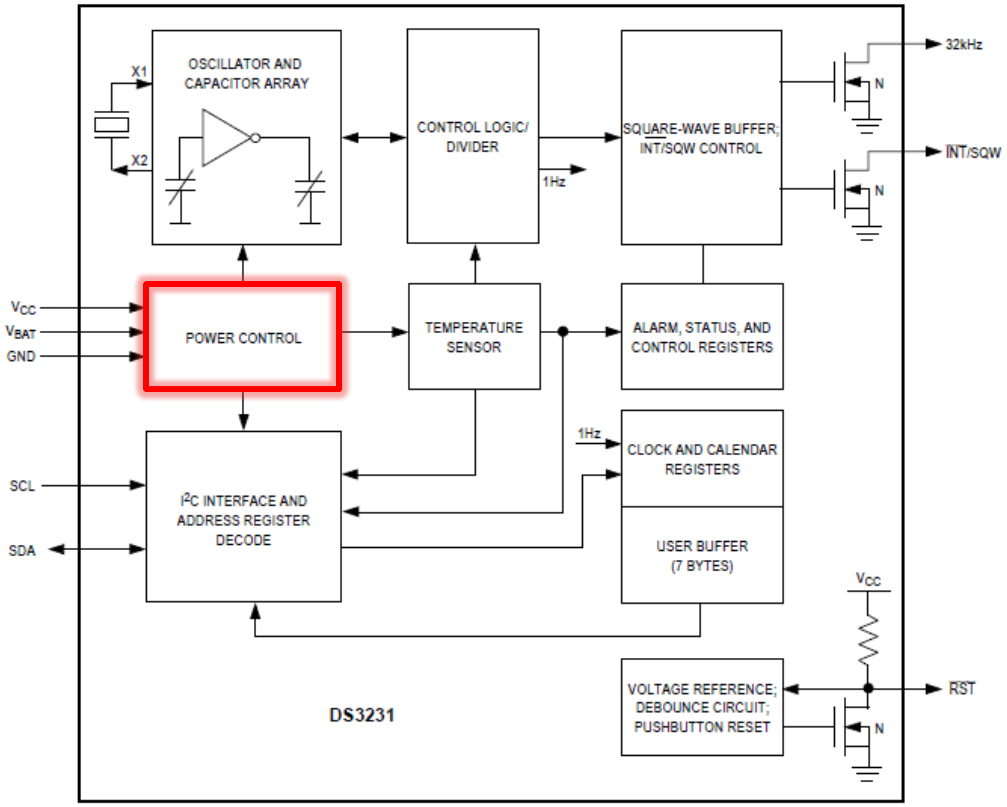

Ilustración 23. Grupo funcional del control de alimentación

Cuando V_{CC} es mayor que V_{PF} (*Power-Fail Voltage* ≈ 2.575 V), el dispositivo se alimenta por medio de la tensión V_{CC}. Lo mismo sucede si V_{CC} es menor que V_{PF} pero mayor que V_{BAT}. Por otra parte, si V_{CC} el menor que V_{PF} y menor que V_{BAT}, el dispositivo se alimenta de la tensión V_{BAT}. Todo esto queda reflejado en la Tabla 2.

Condición de alimentación	Suministro de tensión activo
$V_{CC} < V_{PF}$, $V_{CC} < V_{BAT}$	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}

Tabla 2. Control de alimentación del chip DS3231

Para conservar la batería, la primera vez que se aplica V_{BAT} al dispositivo, el oscilador no empezará a funcionar hasta que V_{CC} supere el valor de V_{PF}, o hasta que una dirección I^2C válida sea escrita al dispositivo. En cuanto eso suceda, el oscilador empezará a funcionar en menos de un segundo. Aproximadamente dos

segundos después de que VCC ha sido aplicada al dispositivo, o de que se escriba una dirección I²C válida, el DS3231 realiza una medida de temperatura y aplica las correcciones oportunas al oscilador. Una vez que el oscilador está funcionando, continúa mientras V_{CC} o V_{BAT} estén disponibles, y el dispositivo continúa con las medidas de temperatura y las correcciones de frecuencia del oscilador cada 64 segundos.

Con la primera aplicación de tensión V_{CC} o V_{BAT}, los registros de hora y fecha se inicializan a 01/01/00 01 00:00:00 (DD/MM/AA DIA-SEMANA HH:MM:SS).

El DS3231 incorpora un pin $\overline{RST}$ para llevar a cabo un reseteo del chip.

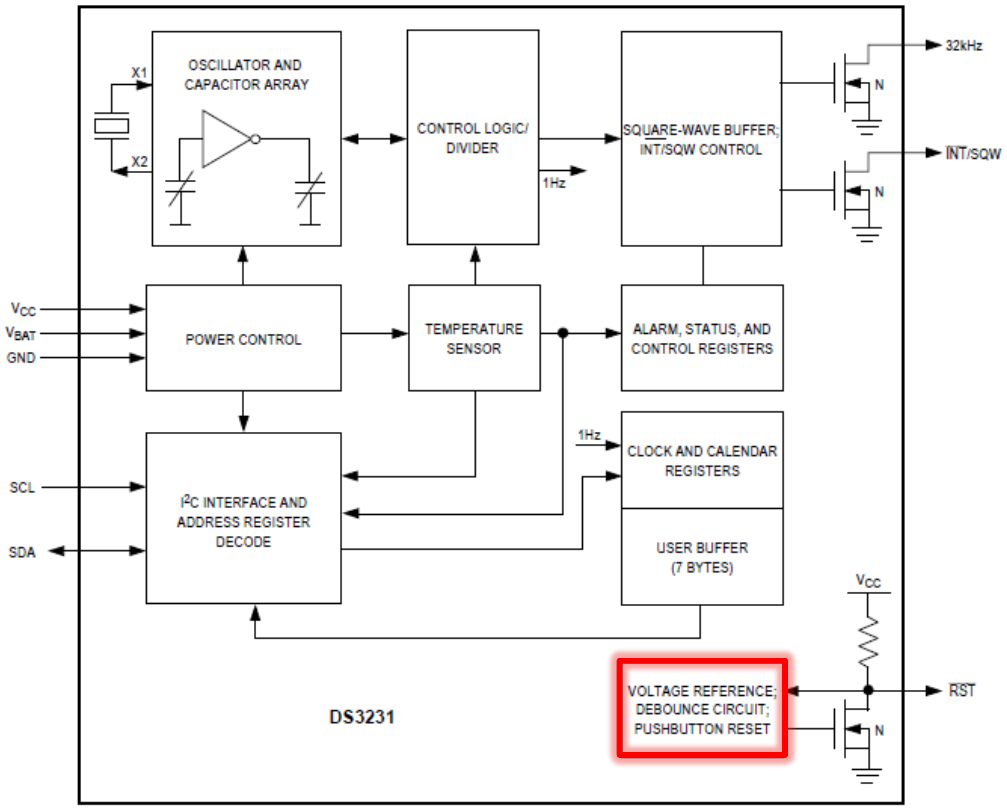

Ilustración 24. Grupo funcional de reseteo del chip

Cuando el DS3231 no se encuentra en un ciclo de reseteo, está continuamente esperando a un flanco de bajada en el pin $\overline{RST}$. Si se detecta una transición de este tipo, el DS3231 activa el interruptor (transistor) colocando el pin a un estado bajo. Después de que el temporizador interno haya finalizado (pushbutton debounce, $PB_{DB} = 250\,ms$), el DS3231 sigue verificando el pin $\overline{RST}$. Si la línea continúa en estado bajo, el DS3231 permanece a la espera de un flaco de subida. Al detectar la liberación en la línea, el DS3231 fuerza el estado del pin

$\overline{RST}$ a un estado bajo durante un tiempo $t_{RST} = 250\ ms$. Todo esto queda recogido en la *Ilustración 25*.

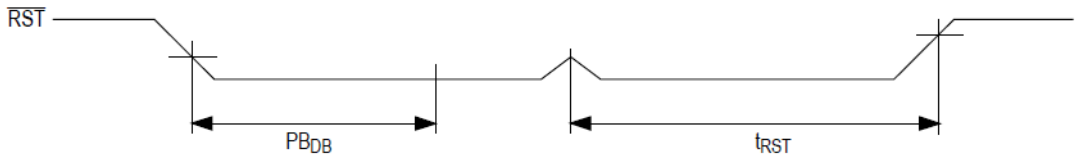

Ilustración 25. Ciclo de reseteo del chip

El pin $\overline{RST}$ también es usado para indicar un fallo en la alimentación. Cuando V_{CC} es menor que V_{PF}, se genera una señal interna para indicar un fallo de alimentación, lo que fuerza a poner el pin $\overline{RST}$ a un estado lógico bajo. Cuando V_{CC} vuelve a ser mayor que V_{PF}, el pin $\overline{RST}$ se mantiene a 0 durante aproximadamente 250 ms (t_{REC}) para permitir estabilizar la señal de alimentación. Si el oscilador no está funcionando cuando se aplica V_{CC} (primera conexión del dispositivo), t_{REC} se omite y $\overline{RST}$ pasa directamente a un estado alto.

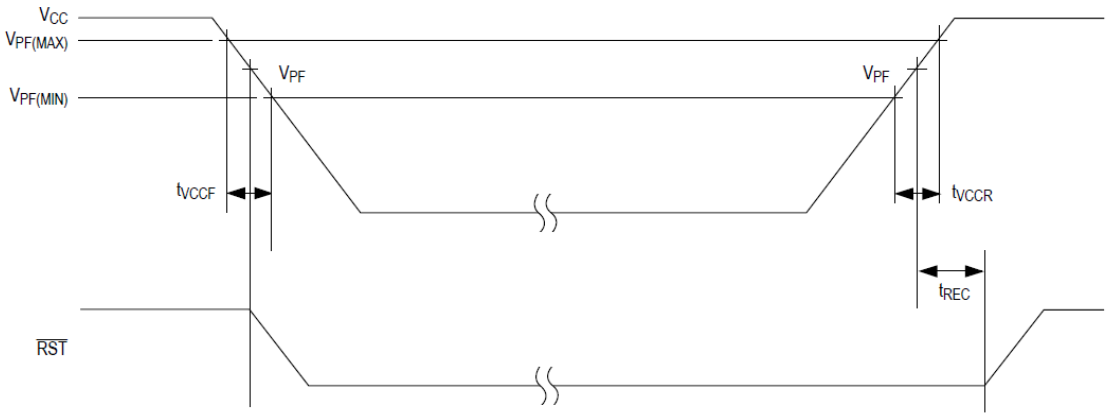

Ilustración 26. Ciclo de detección de fallo de alimentación

Ambas funcionalidades del pin $\overline{RST}$ no afectan al funcionamiento interno del DS3231.

Por último, el bloque funcional del reloj de tiempo real proporciona información actualizada sobre los segundos, minutos, horas, día de la semana, día del mes, mes y año.

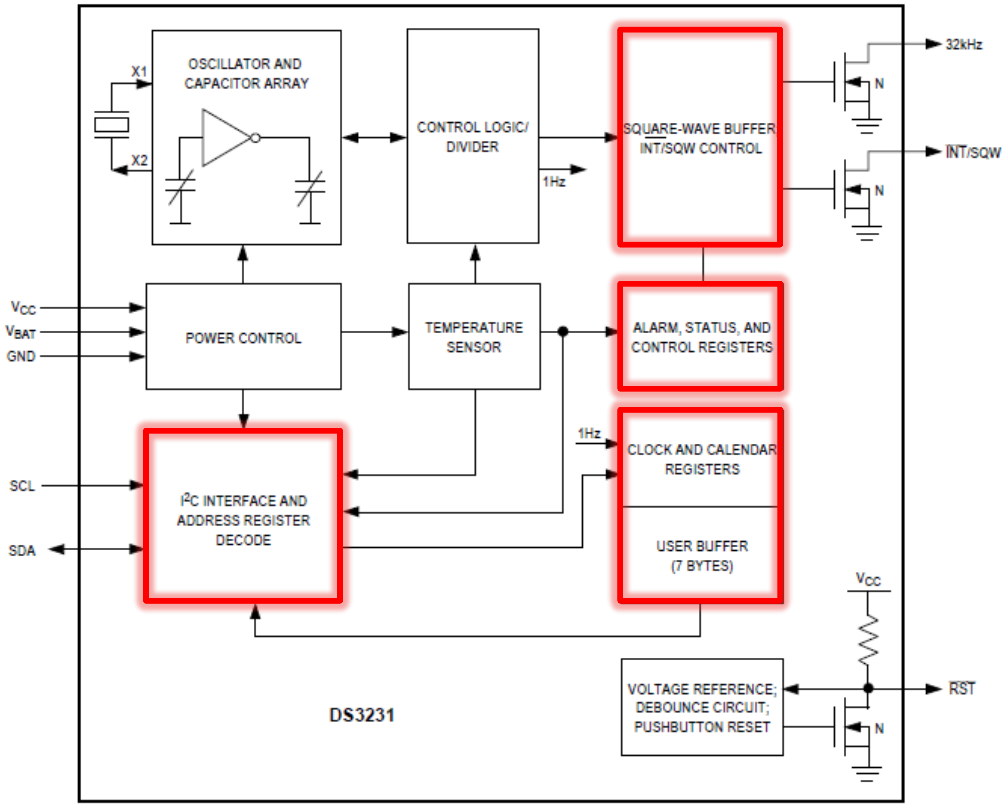

Ilustración 27. Grupo funcional del reloj de tiempo real

El día del mes se ajusta automáticamente cuando finalizan los meses que tienen menos de 31 días, incluyendo las correcciones para los años bisiestos. El reloj trabaja en modo 24 horas, o 12 horas con un bit indicador ($\overline{AM}/PM$) que vale 0 para indicar horario de mañana y 1 para horario de tarde.

El reloj proporciona dos alarmas configurables dentro del mismo día y una señal cuadrada programable. El pin $\overline{INT}/SQW$ desempeña la función, tanto de provocar la interrupción provocada con las alarmas, como la salida de dicha onda cuadrada. La selección de usar ese pin como interrupción de alarmas o como salida de onda cuadrada se controla con un bit del registro de control que se explicará más adelante.

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 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{AM}$/PM 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 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{AM}$/PM 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 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{AM}$/PM 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 3. Direcciones de los registros del DS3231

En la tabla anterior se muestra toda la información de cada uno de los registros del DS3231 (dirección, función, rango de valores…).

2.4.3. Aspectos a tener en cuenta

El reloj de tiempo real DS3231 (el mostrado en la Ilustración 17) está diseñado para llevar una batería de litio recargable modelo LIR2032, por lo que incorpora un circuito de carga de dicha batería cuando el dispositivo está conectado a la alimentación.

El circuito de carga se puede encontrar en el esquema de la Ilustración 28.

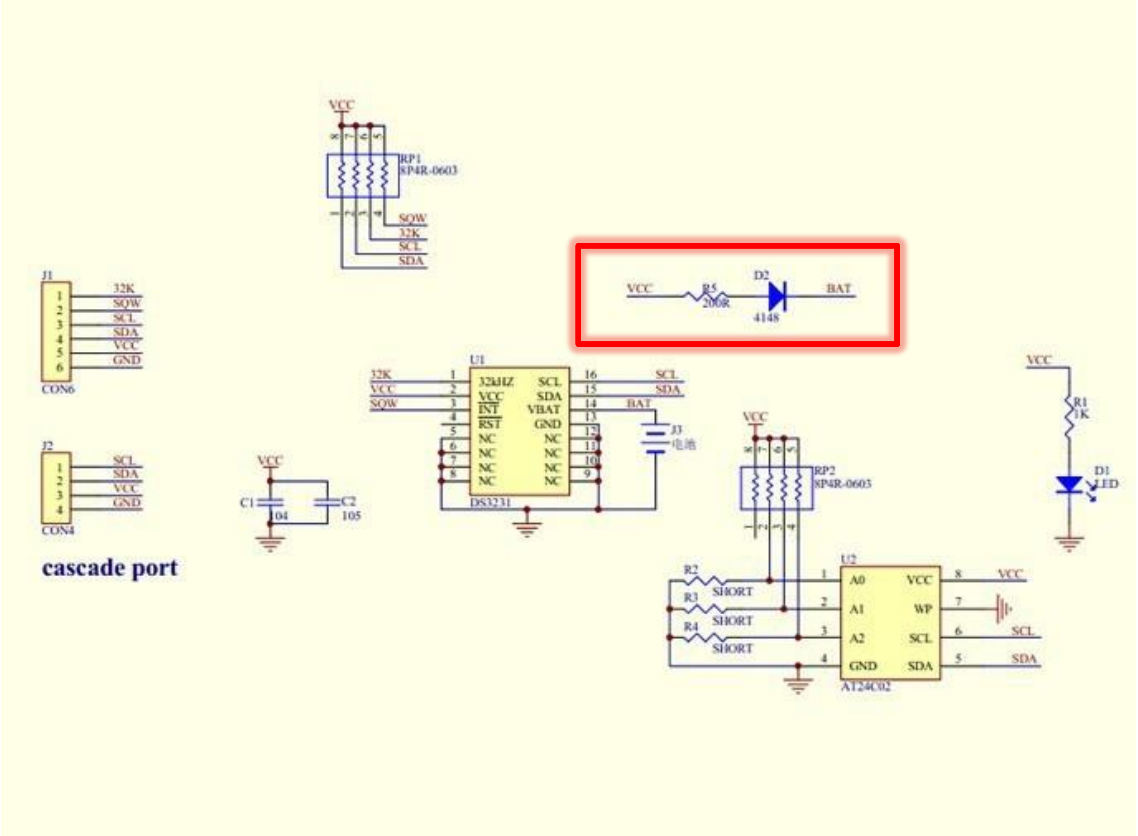

Ilustración 28. Esquema del DS3231

Muchos son los fabricantes que venden el módulo con una pila convencional (CR2032) en lugar de la batería de litio recargable para la que está diseñado. Por este motivo, al adquirir el módulo, es importante comprobar el tipo de batería que trae el dispositivo, en caso de que venga con alguna.

Si el módulo viene con la batería no recargable (CR2032) será necesario sustituirla por el modelo recargable (LIR2032) o, en caso de no ser posible, eliminar el circuito de carga para evitar dañar la pila y con ello el módulo, ya que se puede llegar a sobrecalentar en exceso.

Para eliminar el circuito de carga será necesario desoldar la resistencia R5 y el diodo D2. En la Ilustración 29 se muestra el módulo sin dichos componentes para ver con más claridad dicha modificación.

Ilustración 29. Modificación del DS3231

2.4.4. Conexionado

El diagrama del conexionado del reloj de tiempo real con la Raspberry Pi se muestra en la Ilustración 30.

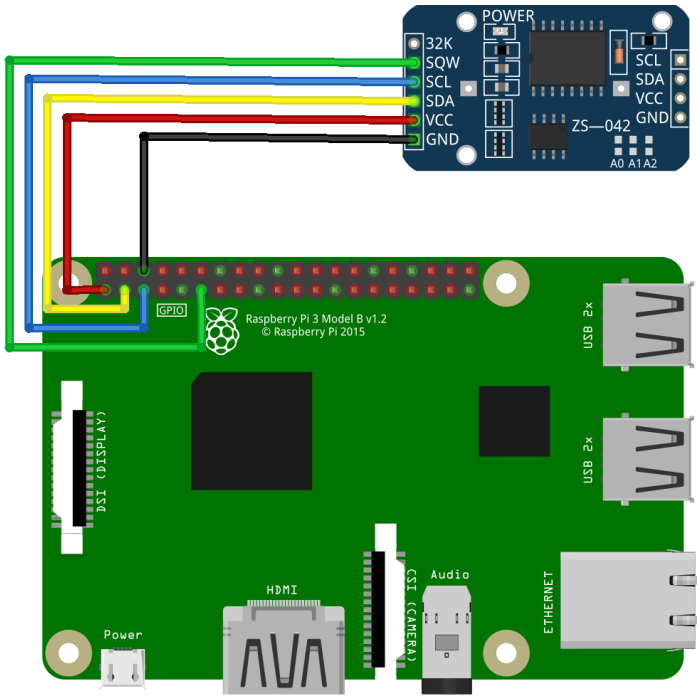

Ilustración 30. Diagrama de conexión

Recordar que según el diagrama que se muestra arriba es importante configurar la resistencia pull-up del pin de entrada de la Raspberry Pi como se mostró en el apartado 2.3.3.

3. DESARROLLO DEL TFG

Lo primero será conectar el módulo RTC a la Raspberry Pi como se mostró en la Ilustración 30 y realizar un escaneo de dispositivos con el comando:

```
i2cdetect -y 1
```

Una vez hecho esto, hay que verificar que aparecen dos direcciones en el muestreo del bus I²C, es decir, que hay dos dispositivos conectados al bus I²C. Estos dos dispositivos son el DS3231 y la memoria EEPROM que incluye el módulo ZS-042, cuyas direcciones son 0x68 y 0x57, respectivamente.

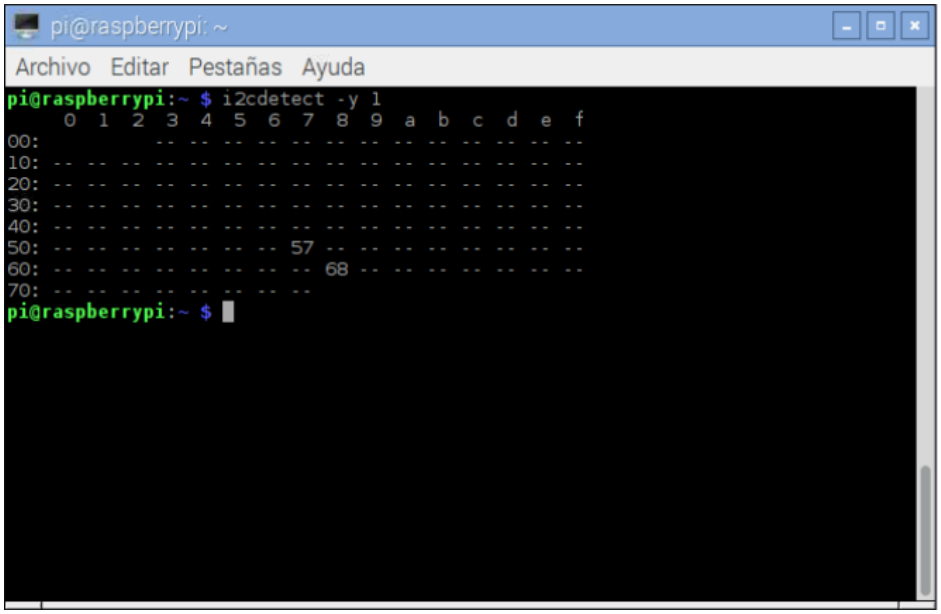

Ilustración 31. Muestreo de direcciones del DS3231 y la memoria EEPROM

Con esto ya se puede empezar a elaborar el código de lo que será el módulo para controlar el DS3231.

Como se mencionó en el apartado anterior, los datos de los registros del DS3231 se almacenan con representación BCD. Es por esto por lo que, para llevar a cabo la escritura de datos en los registros será necesario realizar una previa conversión de decimal a BCD, para evitar almacenar la información de manera errónea. Por el contrario, cuando se quiera leer datos de los registros y trabajar con ellos habrá que realizar la conversión opuesta: de BCD a decimal.

Conociendo todo lo que se explicó sobre definición de funciones en el apartado 2.2.4, se van a crear dos: una que convierta un valor decimal en BCD y otra que convierta un valor BCD a decimal.

La primera de ellas será:

```
def bcd_to_int(bcd):
    n = 0
    for i in (bcd >> 4, bcd):
        for j in (1, 2, 4, 8):
            if i & 1:
                n += j
            i >>= 1
        n *= 10
    return n / 10
```

La función *bcd_to_int* recibe un parámetro: el número binario en representación BCD de 4 bits. La variable *n* es el número decimal de salida, por lo que inicialmente se la inicializará con un valor 0. Lo primero que se hace es dividir el número binario en dos partes: una con los 4 bits más significativos y otra con el número BCD entero. Lo siguiente será ir comparando bit a bit, de derecha a izquierda el número (del bit menos significativo a más significativo), y almacenar en la variable *n* el valor decimal de los "unos" en función de su posición.

Cuando se hayan comparado los 4 bits más significativos, el valor decimal se multiplica por 10 y se comparan los menos significativos. Para esta iteración no se separan los 4 bits menos significativos, sino que se emplea todo el número binario, ya que en el bucle for siguiente sólo se van a hacer 4 iteraciones, correspondientes a esos 4 bits menos significativos.

Una vez acabada la segunda iteración se vuelve a multiplicar el valor decimal por 10, por lo que el valor que devuelve la función será el número decimal final dividido entre 10.

A continuación se va a mostrar un ejemplo para entender mejor el funcionamiento paso a paso de la función:

Se tiene el número binario 01101000, inicialmente n (valor decimal) vale 0.

Lo primero es separar el número en dos partes para las dos iteraciones principales. Estas dos partes son: 0110 y 01101000.
El siguiente bucle for irá comparando bit a bit la primera parte (0110). El primer 1 que aparace está en la segunda posición por lo que n pasa a valer 2. El siguiente está en la tercera posición, se suma a n el valor 4, pasando a valer 6. Se finalizan las 4 comparaciones en busca de bits de valor 1 y se multiplica por 10 el valor de n, pasando a valer 60.

Comienza la segunda iteración principal. El único 1 aparece en la última iteración del segundo bucle for, por que se suma a n el valor 8, ascendiendo a 68 su valor. Se vuelve a multiplicar por 10, pasando n a valer 680.

Acaban todas las iteraciones, por lo que se devuelve el valor decimal n dividido entre 10: 680/10 = 68 (0110 = 6 y 1000 = 8, valor decimal = 68).

Por otra parte, la función que realiza la conversión opuesta es:

```
def int_to_bcd(n):
    bcd = 0
    for i in (n // 10, n % 10):
        for j in (8, 4, 2, 1):
            if i >= j:
                bcd += 1
                i -= j
            bcd <<= 1
    return bcd >> 1
```

En este caso, la función *int_to_bcd* recibe la variable *n*, que será un número decimal, y devolverá su valor en binario con representación BCD almacenado en la variable *bcd*, es por esto por lo que al comienzo de la función se le asigna el valor 0 a dicha variable para inicializarla. De manera análoga a la función *bcd_to_int*, primero se separa el número decimal en dos dígitos (si los tuviera; si no fuera así, el primero será 0).

La primera iteración se realiza con la cifra correspondiente a las decenas y la segunda con la de las unidades. Después se realizan cuatro iteraciones secundarias, en las que se compara la cifra decimal con los valores decimales equivalentes de los "unos" en binario en sentido descendente; es decir, 8, 4, 2 y 1. Empezando así: si la cifra es mayor que ocho, el bit más significativo de los cuatro que se quieren obtener vale 1, se resta ese valor a la cifra (8 en este caso) se almacena ese bit y se realiza la siguiente iteración comparando con el 4. Si no, pasa directamente a la iteración del 4, y ese bit será 0.

Como en la última iteración se almacena el bit, y para ello se desplaza todo un bit hacia izquierda, el valor que devuelve la función tendrá que desplazarse un bit hacia la derecha.

Para entenderlo un poco mejor se va a tratar el ejemplo de la función *bcd_to_int* de manera inversa:

Se tiene el número decimal 68, bcd inicialmente vale 0.

Lo primero es separar el número en sus dos decimales: el de las decenas y el de las unidades, es decir, 6 y 8. Se realizarán dos iteraciones principales, una para el 6 y otra para el 8.

En la primera iteración se compara en el segundo bucle for el 6 con el 8. Al ser menor, el primer bit más significativo valdrá 0. En la segunda iteración secundaria se compara con el 4 y, como es mayor, el segundo bit más significativo valdrá 1. Se resta 4 a 6, valiendo ahora la variable i = 2.

En la siguiente iteración al ser 2 = 2, el tercer bit valdrá uno también, pasando i a valer 0. Por lo que el cuarto bit vale 0. Como resultado se obtienen los 4 primeros bits de la conversión, quedando 0110. Se realiza lo mismo para el siguiente dígito, el 8, quedando 1000.

Acaban todas las iteraciones, por lo que se devuelve el número binario con representación BCD resultante: 01101000 (0110 = 6, 1000 = 8).

Estas dos funciones estarán incluidas en el módulo objetivo de este trabajo.

Pero, antes de nada, todos los scripts que se creen deberán llevar una cabecera que será:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-
```

La finalidad de la primera línea de la cabecera es la de indicar la ruta en la que se encuentra el intérprete de Python para que, cuando se ejecute el programa desde el terminal, el sistema sepa la ruta en la que se encuentra el intérprete para dicho script.

La segunda indica el tipo de codificación del script. Por defecto la codificación es la utf-8 pero, en este caso, se usa la ISO-8859-15. Esto permite que se puedan utilizar caracteres especiales del castellano como pueden ser las tildes o la letra *ñ*, evitando los problemas en la ejecución que surgirían si se utilizase la codificación utf-8.

Además, es necesario exportar los módulos que contienen las funciones con las que se va a trabajar. Estos módulos son *RPi.GPIO*, *datetime* y *smbus*, por lo que la cabecera del módulo quedará:

```
import RPi.GPIO as GPIO
from datetime import datetime
from smbus import SMBus
```

También se definirá una constante para indicar el siglo actual. El objetivo de definirla al comienzo del programa es el de no tener que buscar en el script dónde se emplea el valor del siglo actual; es más sencillo disponer de este valor al comienzo del mismo y poder modificarlo cómodamente. Más adelante se explicará la finalidad:

```
SIGLO = 21
```

Lo siguiente será definir dentro de este módulo la clase DS3231, que contendrá todas las funciones para controlar y programar el RTC. La cabecera para la definición de dicha clase será:

```
class DS3231():
```

Todos los métodos que pertenezcan a esta clase deberán ir tabulados en el código del módulo.

Dentro de la clase, lo primero en definir serán los registros que se mostraron en la Tabla 3 para que, más adelante, se trabaje de manera más cómoda y visual.

```
REG_SEGUNDOS = 0x00
REG_MINUTOS = 0x01
REG_HORA = 0x02
REG_DIA_SEMANA = 0x03
REG_DIA = 0x04
REG_MES = 0x05
REG_ANIO = 0x06
REG_A1_SEGUNDOS = 0x07
REG_A1_MINUTOS = 0x08
REG_A1_HORA = 0x09
REG_A1_DIA = 0x0a
REG_A2_MINUTOS = 0x0b
REG_A2_HORA = 0x0c
REG_A2_DIA = 0x0d
REG_CONTROL = 0x0e
REG_STATUS = 0x0f
REG_TEMP_MSB = 0x11
REG_TEMP_LSB = 0x12
```

El primer método que debe aparecer dentro de la clase es el de inicialización o constructor. Será el encargado de crear el canal de comunicación I²C. Deberá nombrarse "*__init__*" y su ejecución se realiza de manera automática.

```
def __init__(self, direccion=0x68, at24c32_direccion=0x57):
        if GPIO.RPI_REVISION == 1:
            i2c_bus = 0
        elif GPIO.RPI_REVISION == 2:
            i2c_bus = 1
        elif GPIO.RPI_REVISION == 3:
            i2c_bus = 1
        else:
            raise ValueError("No ha sido posible determinar la
                              version de la Raspberry Pi" +
                             "\nNo ha sido posible establecer
                              la comunicacion con el RTC")

        print("Creando bus I2C...")
        self.bus = SMBus(i2c_bus)
        self.direccion = direccion
        self.at24c32_direccion = at24c32_direccion
        print("Bus I2C creado")
```

El primer parámetro del método es *self*. *self* simplemente es un referencia al objeto que se asocia a la clase, es por esto por lo que aparecerá en todos los

métodos que pertenecen a la clase. Su uso es similar al de "this" empleado en C++ o JavaScript.

Esto se entiende mejor con dos ejemplos:

Teniendo el método perteneciente a la clase DS3231 definida como:

```
def escribir (self, registro)
```

Sería lo mismo: `DS3231.escribir(objeto)`

Que:

```
objeto = DS3231()
objeto.escribir()
```

Es decir, la llamada *objeto.metodo(parametro)* se traduce como *clase_del_objeto(objeto, parametro)*.

```
1.    class Test(object):
2.          def method_one(self):
3.                print "Called method_one"
4.
5.          def method_two():
6.                print "Called method_two"
7.
8.    a_test = Test()
9.    a_test.method_one()
10.   a_test.method_two()
```

En este caso, las líneas 8 y 9 se traducirían como *Test.method_one(a_test)* mientras que la línea 10 fallaría con un error *Typedef*.

(*Class method differences in Python: bound, unbound and static*, 2008. Recuperado de https://stackoverflow.com/questions/114214/class-method-differences-in-python-bound-unbound-and-static)

Los otros dos parámetros del método constructor son las direcciones del RTC y de la memoria EEPROM, respectivamente. La definición de los parámetros del modo que se muestra permite que, en caso de que no se pasen valores distintos, las direcciones por defecto son 0x68 y 0x57, como se mencionó anteriormente.

Lo primero que aparece en el método es un bucle if para definir la versión de la Raspberry para la creación del bus I²C de comunicación y que, de este modo, sirva para los diversos modelos de Raspberry Pi, sin necesidad de cambiar manualmente el módulo cuando se trabaje con una versión distinta de Raspberry Pi. Esta es la finalidad de exportar el módulo *RPi.GPIO*.

Una vez obtenido el valor de la variable i2c_bus se procede a crear el bus I²C a partir de dicho valor y se almacenan en las variables *self.direccion* y *self.at24c32_direccion* las direcciones del RTC y de la memoria EEPROM,

respectivamente. De este modo, podrán ser usadas por el resto de funciones de la clase.

3.1. Métodos elementales de lectura y escritura

Ahora se van a realizar los métodos "elementales" para escribir y leer en los registros. Dichos métodos se muestran a continuación:

```
def escribir(self, registro, dato):
    self.bus.write_byte_data(self.direccion, registro, dato)
```

El método para escribir en los registros tiene tres parámetros: *self*, *registro* y *dato*. *self*, como se explicó arriba, aparecerá como parámetro en todas las funciones del módulo. La función SMBus que se utiliza es *write_byte_data* y su finalidad es escribir "dato" en el registro interno "registro" del dispositivo que tiene como dirección "self.direccion".

Por otra parte, la función para leer los registros será:

```
def leer(self, registro):
    return self.bus.read_byte_data(self.direccion, registro)
```

En este caso, la función de lectura tiene dos registros: *self* y *registro*, y la función SMBus empleada es *read_byte_data*. Esta función realizará la lectura del registro interno "registro" del dispositivo del bus I^2C que tiene como dirección "self.direccion". La función devuelve el dato obtenido en la lectura.

Ambas funciones, *escribir* y *leer*, son a las que realizarán llamadas las funciones más específicas que se mostrarán más adelante ya que, como se ha visto hasta ahora, no se hace uso de las funciones de conversión de decimal a BCD, y viceversa. Serán estas funciones más específicas las que hagan uso de ellas.

Por lo tanto, el módulo nombrado como *DS3231.py* queda hasta ahora del modo que se muestra a continuación:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                                DS3231.py                                   #
#                                                                            #
# Módulo para controlar el chip DS3231 con distintas funciones de usuario.   #
#                                                                            #
##############################################################################

import RPi.GPIO as GPIO
from datetime import datetime
from smbus import SMBus
import time
```

```
# Definicion del siglo actual
SIGLO = 21

# Transforma un número binario (BCD 2x4bits) en uno entero
def bcd_to_int(bcd):
    n = 0
    for i in (bcd >> 4, bcd):
        for j in (1, 2, 4, 8):
            if i & 1:
                n += j
            i >>= 1
        n *= 10
    return n / 10

# Transforma un numero entero en uno binario (BCD 2x4bits)
def int_to_bcd(n):
    bcd = 0
    for i in (n // 10, n % 10):
        for j in (8, 4, 2, 1):
            if i >= j:
                bcd += 1
                i -= j
            bcd <<= 1
    return bcd >> 1

# Clase que contendra cada uno de los registros  y funciones del DS3231
class DS3231():
# Definicion de los registros
    REG_SEGUNDOS = 0x00
    REG_MINUTOS = 0x01
    REG_HORA = 0x02
    REG_DIA_SEMANA = 0x03
    REG_DIA = 0x04
    REG_MES = 0x05
    REG_ANIO = 0x06
    REG_A1_SEGUNDOS = 0x07
    REG_A1_MINUTOS = 0x08
    REG_A1_HORA = 0x09
    REG_A1_DIA = 0x0a
    REG_A2_MINUTOS = 0x0b
    REG_A2_HORA = 0x0c
    REG_A2_DIA = 0x0d
    REG_CONTROL = 0x0e
    REG_STATUS = 0x0f
    REG_TEMP_MSB = 0x11
    REG_TEMP_LSB = 0x12

################################################################################
#                                                                              #
#                                    INIT                                      #
#                                                                              #
################################################################################

    def __init__(self, direccion=0x68, at24c32_direccion=0x57):
        #Detecta automaticamente la version de la RPi para crear el bus I2C
        if GPIO.RPI_REVISION == 1:
            i2c_bus = 0
        elif GPIO.RPI_REVISION == 2:
            i2c_bus = 1
        elif GPIO.RPI_REVISION == 3:
            i2c_bus = 1
        else:
            raise ValueError("No ha sido posible determinar la version " +
                             "de la Raspberry Pi" +
                             "\nNo ha sido posible establecer la " +
                             "comunicacion con el RTC")
```

```
        # print("Version de la Raspberry Pi: " + str(GPIO.RPI_REVISION))
        print("Creando bus I2C...")
        self.bus = SMBus(i2c_bus)
        self.direccion = direccion
        self.at24c32_direccion = at24c32_direccion
        print("Bus I2C creado")

#############################################################################
#                                                                           #
#                                ESCRIBIR                                   #
#                                                                           #
#############################################################################

    # Funcion primitiva para escritura de un dato en un registro
    def escribir(self, registro, dato):
        self.bus.write_byte_data(self.direccion, registro, dato)

#############################################################################
#                                                                           #
#                                  LEER                                     #
#                                                                           #
#############################################################################

    # Funcion primitiva para lectura de registro a traves del bus
    def leer(self, registro):
        return self.bus.read_byte_data(self.direccion, registro)
```

Ahora se van a realizar dos scripts de prueba para comprobar el funcionamiento de todo lo visto hasta ahora, es decir, establecer la comunicación entre la Raspberry Pi y el módulo RTC a través del bus I^2C y la lectura y escritura en los registros del DS3231.

El primero de ellos va a realizar la escritura en un registro elegido por el usuario de un dato decimal elegido también por este. Este script se muestra a continuación:

***NOTA**: todos los ficheros de prueba que se muestran se incluyen en el ANEXO.*

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

#############################################################################
#                                                                           #
#                           escribir_registro.py                            #
#                                                                           #
# Programa que solicita un registro y un valor decimal al usuario para      #
# realizar su escritura en dicho registro.                                  #
#                                                                           #
#############################################################################

import DS3231

RTC = DS3231.DS3231()

reg = input("\n\tIntroduzca registro: ")
dato = input("\tIntroduzca un valor decimal: ")

RTC.escribir(reg, DS3231.int_to_bcd(dato))
```

Comentando un poco el funcionamiento del script, lo primero que hace es importar el módulo *DS3231.py* que se había creado anteriormente y se crea el objeto *RTC* asignándole a la clase *DS3231* para poder hacer uso de todas las funciones que esta incluye; es decir, las funciones *escribir* y *leer*. Lo siguiente es solicitar al usuario un registro donde escribir y el dato a escribir, almacenando esos valores en las variables *reg* y *dato*, respectivamente.

Con toda esta información se realiza la llamada a la función *escribir* pasando como parámetros *reg* y *dato*, pero esta última realizada la conversión de decimal a BCD mediante la llamada a la función *int_to_bcd* como se muestra en el script.

Para probar este script se realizará una escritura en el registro que contiene el valor del año actual; es decir, el que tiene la dirección 0x06 (ver Tabla 3). Se ha elegido este para que, cuando se compruebe que la escritura es correcta con una lectura posterior del registro, el valor no haya variado, ya que la lectura se realiza en el mismo año. Además, este registro es el que tiene un rango más amplio de valores (0-99).

También se puede realizar la escritura en cualquier otro registro como, por ejemplo, el que almacena los minutos (0x01) para comprobar que, si se realiza la lectura un tiempo después de la escritura, ese valor se ha incrementado a razón +1 cada minuto.

La ejecución del script anterior se efectúa mediante el uso del terminal con el siguiente comando:

```
sudo python escribir_registro.py
```

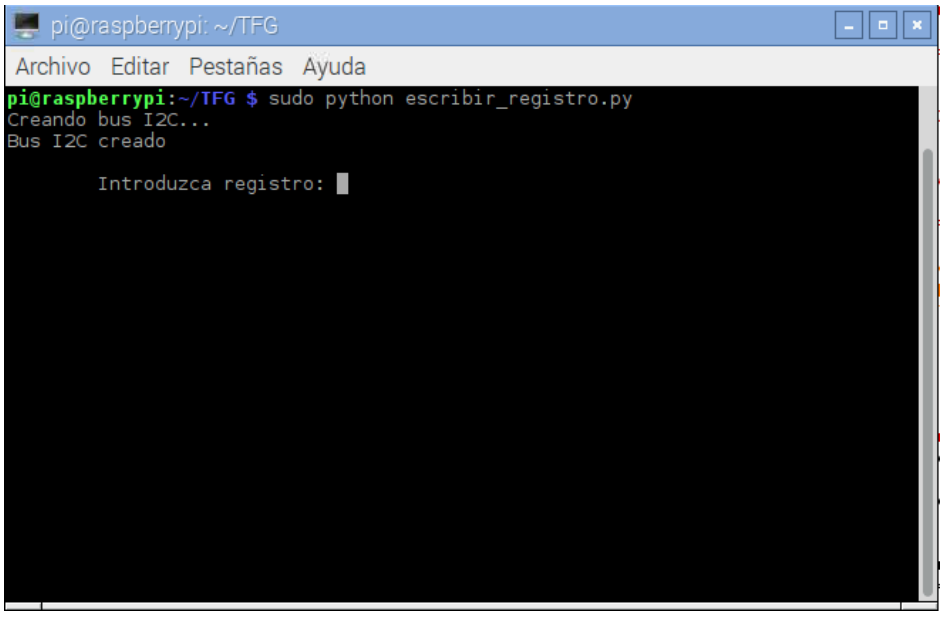

Ilustración 32. Ejecución del script *escribir_registro.py*

El resultado sería el inicio del programa creado, con la visualización del mensaje que solicita la introducción del registro en el que se desea realizar la escritura. Como se mencionó antes, se introducirá el *0x06* y se pulsa *Enter*.

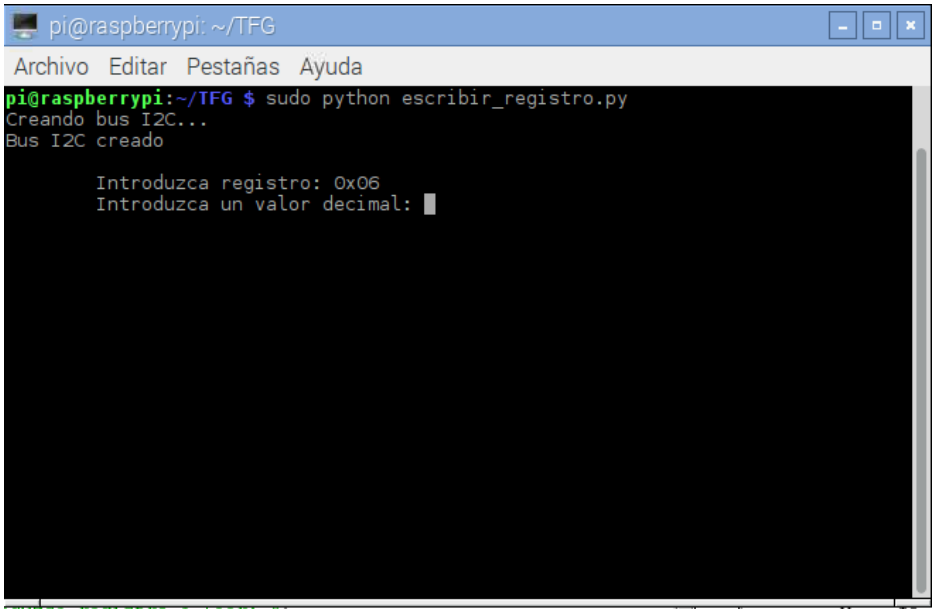

Ilustración 33. Introducción de la dirección del registro

Hecho esto, aparecerá el siguiente mensaje, el que solicita el dato decimal que se quiere escribir en el registro. Recordar que el rango de valores es entre 0 y 99. Se va a introducir un valor cualquiera; por ejemplo, el 68.

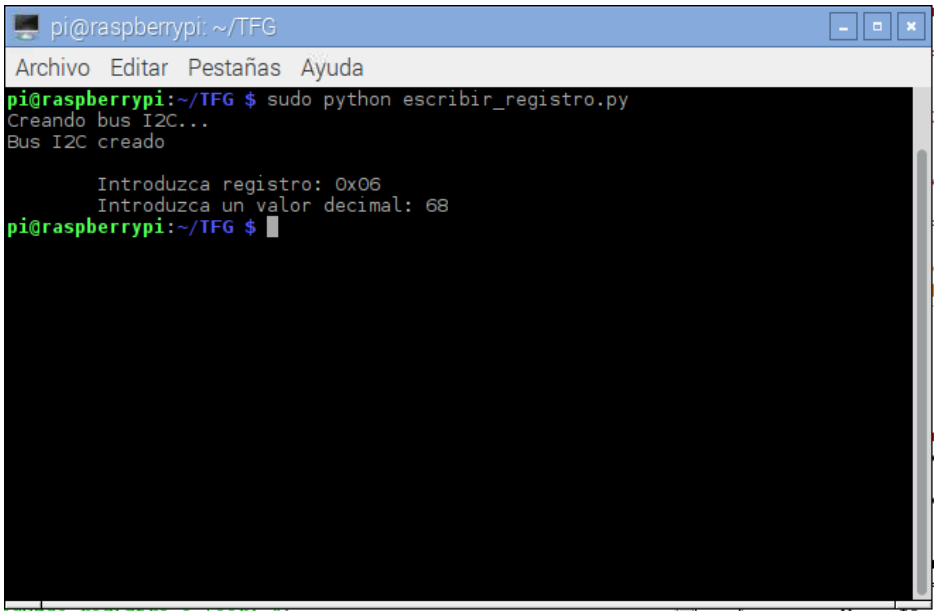

Ilustración 34. Introducción del valor decimal

Al pulsar la tecla *Enter* para introducir el dato, se realiza la escritura y finaliza la ejecución del programa.

Ahora se va a realizar el script que permita leer un registro. La manera de operar será la misma; se solicitará por pantalla la introducción manual de la dirección del registro deseado y, posteriormente, se mostrará el valor que contiene dicho registro. Este valor se mostrará de dos maneras: una con el valor sin convertir; es decir, en binario con representación BCD, y la otra con la conversión de BCD a decimal mediante el uso de la función *bcd_to_int* Esto es un modo de comprobar que las funciones de conversión funcionan de manera correcta.

El script de lectura de registros queda:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                            leer_registro.py                                #
#                                                                            #
# Programa que solicita un registro del que se desea realizar la lectura,    #
# obteniendo el valor binario del registro y su conversión a un valor        #
# decimal                                                                    #
#                                                                            #
##############################################################################

import DS3231

RTC = DS3231.DS3231()

reg = input("\n\tIntroduzca registro a leer: ")

dato = RTC.leer(reg)
print("\n\tEl byte del registro 0x%x es: " % (reg) + bin(dato))
print("\tEl valor del registro en decimal es: " +
      str(DS3231.bcd_to_int(dato)) + "\n")
```

La ejecución del script se lleva a cabo de un modo similar al anterior:

```
sudo python leer_registro.py
```

Al iniciarse la ejecución, se muestra un mensaje que solicita la introducción del registro del que se quiere realizar la lectura. Para continuar con el ejemplo, se introducirá el 0x06 para comprobar que el valor que se introdujo es el correcto (68).

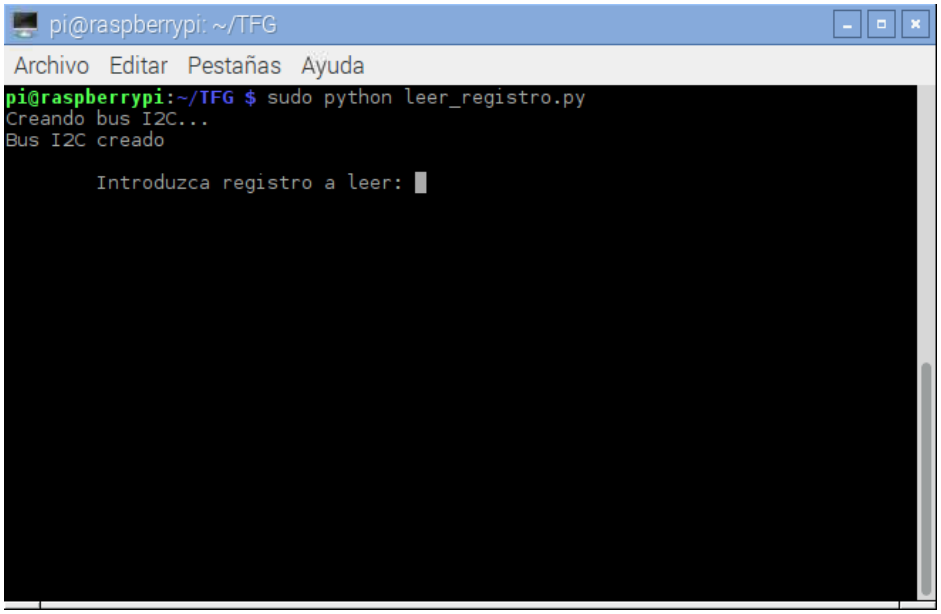

Ilustración 35. Ejecución del script *leer_registro.py*

Tras introducir el valor 0x06 del registro se muestra lo siguiente:

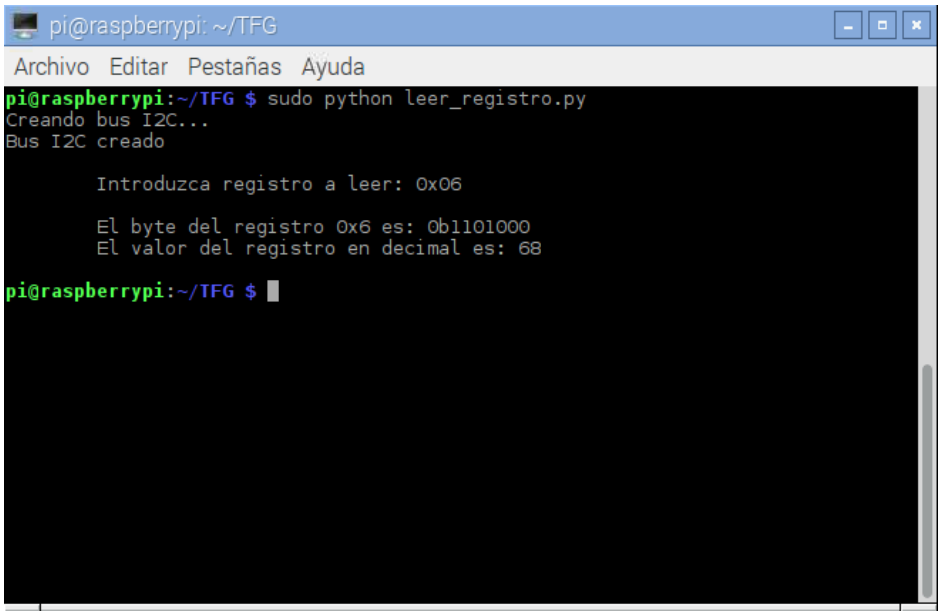

Ilustración 36. Lectura del registro

Como se observa, el valor que contiene el registro es el correcto (68) y, si se realiza la conversión, el valor binario 01101000 se corresponde con ese valor decimal.

3.2. Planteamiento de las funcionalidades del módulo

Una vez que se ha comprobado que la comunicación a través del bus I2C es satisfactoria, y que la escritura y lectura de registros es correcta, se va a proceder a realizar funciones más específicas, como actualizar los datos de fecha y hora, modificarlos, crear alarmas, entre otras.

Para ello, lo primero que se va a hacer es establecer una serie de funcionalidades que el módulo a crear debería cumplir. Estas son:

- Introducción manual de datos para la actualización de la fecha y de la hora.
- Actualización automática de los datos de fecha y hora con la hora actual de la Raspberry Pi (hora sincronizada con internet).
- Realizar lectura de cada uno de los registros (segundos, minutos, horas…) así como de una lectura conjunta de todos ellos.
- Obtener la lectura de los registros de temperatura para poder proporcionar un valor en grados Celsius.
- Creación de alarmas que provoquen una interrupción al ser activadas. Dentro de todas las posibilidades que ofrece el DS3231 (ver apartado 2.4.2), se van a crean 3 modos de funcionamiento de cada una de las dos alarmas:
    - Modo 1: Alarma que se activa a la hora, minutos asignados, sin importar el día.
    - Modo 2: Alarma que se activa a la hora, minutos y día del mes asignados.
    - Modo 3: Alarma que se activa a la hora, minutos y día de la semana asignados.
- Poder obtener información de las alarmas creadas, saber cuáles están activas, pudiendo activarlas o desactivarlas si se desea.

3.3. Funciones de usuario

A partir de todas las características iniciales propuestas en el apartado anterior, se van a desarrollar una serie de funciones de usuario (métodos) que permitan desempeñar cada una de las funcionalidades anteriores. Para ello, se hará uso de los métodos elementales creados inicialmente en el módulo.

3.3.1. Actualizar datos de fecha y hora

La primera funcionalidad de la que dispondrá el usuario es la de inicializar los valores de fecha y hora; es decir, escribir en los registros los valores actuales de las mismas.

Para ello, se va a disponer de modos de actualización de estos datos: manual y automático.

En el primero de ellos será necesario pasar cada uno de los parámetros manualmente al método nombrado como *escribir_fecha*. La definición de dicho método se muestra a continuación:

```
def escribir_fecha(self, hora, minutos, segundos, dia_semana, dia_mes,
                   mes, anio):
        if hora is not None:
            if hora < 0 or hora > 23:
                raise ValueError("Horas fuera de rango [0,23]")
            self.escribir(self.REG_HORA, int_to_bcd(hora))

        if minutos is not None:
            if minutos < 0 or minutos > 59:
                raise ValueError("Minutos fuera de rango [0,59]")
            self.escribir(self.REG_MINUTOS, int_to_bcd(minutos))

        if segundos is not None:
            if segundos < 0 or segundos > 59:
                raise ValueError("Segundos fuera de rango [0,59]")
            self.escribir(self.REG_SEGUNDOS, int_to_bcd(segundos))

        if dia_semana is not None:
            if dia_semana < 1 or dia_semana > 7:
                raise ValueError("Dia de la semana fuera de rango [1,7]")
            self.escribir(self.REG_DIA_SEMANA, int_to_bcd(dia_semana))

        if dia_mes is not None:
            if dia_mes < 1 or dia_mes > 31:
                raise ValueError("Dia del mes fuera de rango [1,31]")
            self.escribir(self.REG_DIA, int_to_bcd(dia_mes))

        if mes is not None:
            if mes < 1 or mes > 12:
                raise ValueError("Mes fuera de rango [1,12]")
            self.escribir(self.REG_MES, int_to_bcd(mes))

        if anio is not None:
            if anio < 0 or anio > 99:
                raise ValueError("Anio fuera de rango [0,99]")
            self.escribir(self.REG_ANIO, int_to_bcd(anio))
```

El método comprobará que cada uno de los parámetros introducidos está dentro de su rango de valores específico. Si el valor cumple la condición de ser un dato válido, se hace una llamada al método elemental *escribir* pasando el dato convertido a binario con representación BCD para su escritura en el registro correspondiente a cada uno de ellos.

Si no se quisiera modificar algún valor, el parámetro que se deberá enviar a la función en lugar del valor será “None”. De ahí que la primera comprobación sea determinar que el valor introducido no sea “None”.

Destacar que el modo de trabajo de las horas es en formato 24h; factor a tener en cuenta a la hora de introducir este dato para evitar errores posteriores.

El día de la semana es un valor numérico entero entre 1 y 7, siendo 1 el lunes y 7 el domingo. El mes será un valor numérico comprendido entre 1 y 12;

enero será el 1 y diciembre el 12. Además, el parámetro “anio” se corresponde con las dos últimas cifras del año; es decir, 17 será el año 2017 (SIGLO = 21).

Para probar el funcionamiento de este método, se ha creado un script de prueba que solicita por pantalla cada uno de los parámetros del método *escribir_fecha* para, posteriormente, realizar una llamada al mismo con todos ellos.

Dicho script *actualizar_manual.py* se muestra a continuación:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                             actualizar_manual.py                           #
#                                                                            #
# Programa para actualizar los valores de fecha y hora. Se solicita al       #
# usuario que introduzca todos los valores manualmente.                      #
#                                                                            #
##############################################################################

import DS3231

RTC = DS3231.DS3231()

# Mensaje inicial
print("\n\tACTUALIZACION DE DATOS DE HORA Y FECHA")
print("\tSi no se desea modificar algun dato, introducir None")

# Se solicitan los datos por pantalla
hora = input("\n\tIntroduzca la hora (formato 24h): ")
minutos = input("\tIntroduzca los minutos: ")
dia_semana = input("\tIntroduzca el dia de la semana (1-7): ")
dia_mes = input("\tIntroduzca el dia del mes: ")
mes = input("\tIntroduzca mes (1-12): ")
anio = input("\tIntroduzca anio (aa): ")


# (hora, minutos, segundos, dia_semana, dia_mes, mes, anio(aa))
RTC.escribir_fecha(hora, minutos, 0, dia_semana, dia_mes, mes, anio)

print("\n\tDATOS DE HORA Y FECHA ACTUALIZADOS\n")
```

Como se observa, se van almacenando cada uno de los datos en variables, que serán las que se usarán como parámetros al hacer la llamada al método *escribir_fecha*.

Como el script muestra por pantalla, si no se quisiera modificar algún dato, bastará con escribir “None” cuando se pida dicho dato.

Al acabar la escritura de los datos se muestra un mensaje por pantalla y finaliza la ejecución del programa.

Para realizar la ejecución del script anterior, bastará con abrir el terminal, situarse en el directorio en el que se encuentra el fichero .py y escribir:

```
sudo python actualizar_manual.py
```

Al pulsar Enter iniciará la ejecución del programa como se ve en la siguiente ilustración:

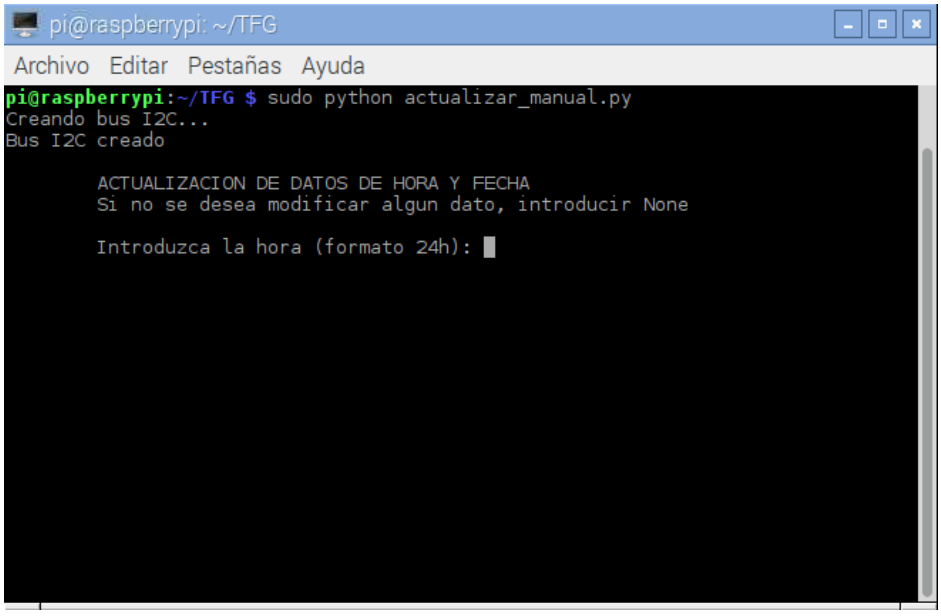

Ilustración 37. Ejecución del script *actualizar_manual.py*

Ahora bastará con ir introduciendo los valores numéricos correspondientes y pulsar Enter, hasta que todos hayan sido introducidos.

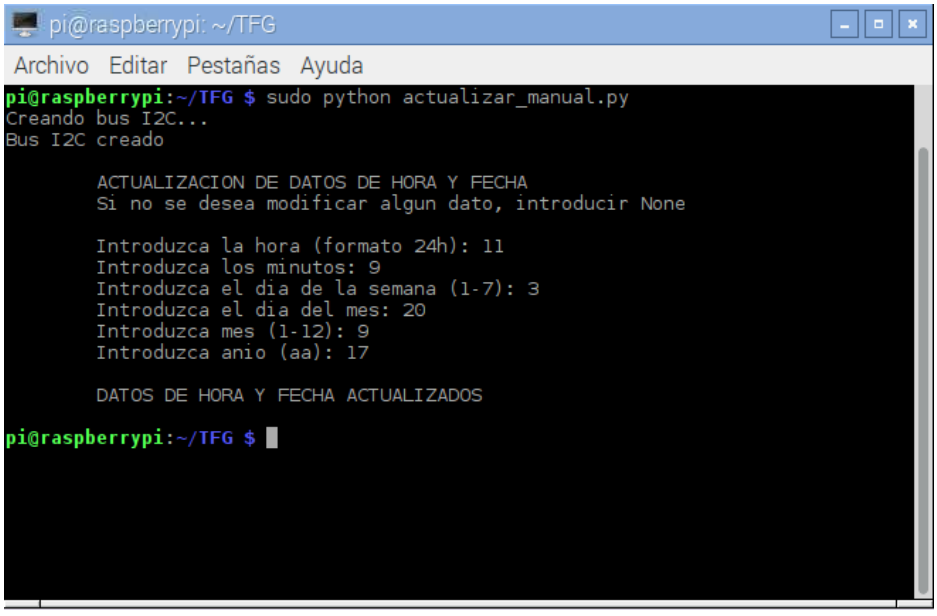

Ilustración 38. Actualización manual de los datos de hora y fecha

El siguiente modo de actualización de hora y fecha es el automático. Se va a crear un método que toma los valores de hora y fecha que tiene la Raspberry Pi y hace una llamada al método anterior (*escribir_fecha*) para almacenarlos.

Es importante que la Raspberry Pi disponga de conexión a internet al utilizar este modo de actualización para que los datos de los que disponga la Raspberry Pi estén actualizados.

La función que realiza la lectura de estos valores es *now()*, perteneciente al módulo *datetime*; es por esto por lo que se importó al comienzo. Esta función devuelve una variable que contiene cada uno de los parámetros que se desean obtener y con ello, poder hacer la llamada al método *escribir_fecha*.

Este método, denominado "actualizar_fecha" es:

```
def actualizar_fecha(self):
        dt = datetime.now()
        self.escribir_fecha(dt.hour, dt.minute, dt.second, dt.isoweekday(),
                            dt.day, dt.month, dt.year % 100)
```

Nuevamente, se ha realizado un script que permita hacer uso de este método y así, poder actualizar los registros del DS3231 de manera automática.

Dicho script, denominado "actualizar_auto", se muestra a continuación:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                            actualizar_auto.py                              #
#                                                                            #
# Programa para actualizar de manera automática los valores de fecha y hora  #
# a través de la función datetime.now(). Es necesario que la Raspberry       #
# disponga de conexión a internet para que la hora esté sincronizada         #
#                                                                            #
##############################################################################

import DS3231

RTC = DS3231.DS3231()

# Inicialización de la hora y fecha con las de la RPi (datetime)
RTC.actualizar_fecha()

print("\n\tDATOS DE HORA Y FECHA ACTUALIZADOS\n")
```

El script, como se muestra, únicamente hace una llamada al método, muestra un mensaje por pantalla para indicar que los datos se han actualizado y finaliza su ejecución.

La ejecución del programa se lleva a cabo con:

```
sudo python actualizar_auto.py
```

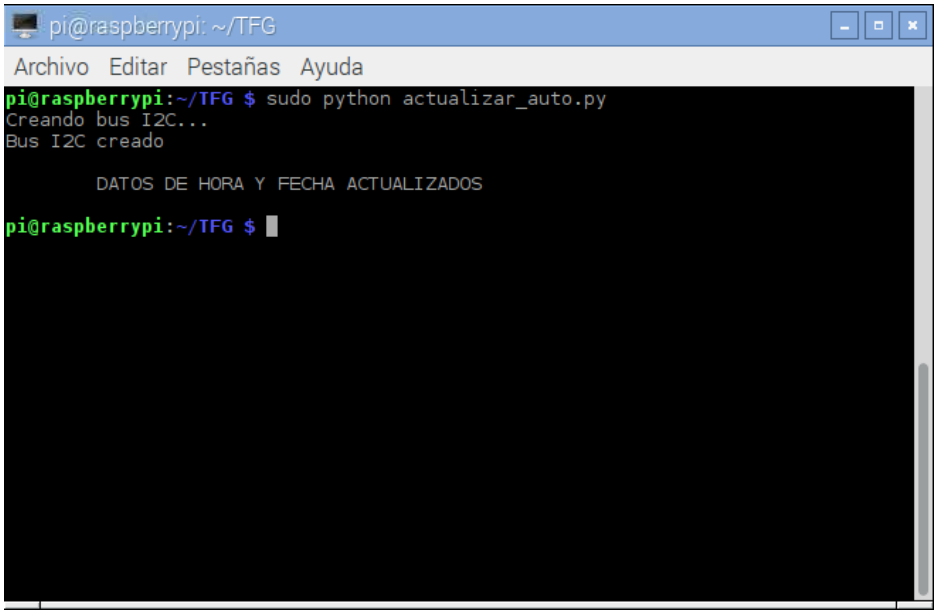

Ilustración 39. Ejecución del script *actualizar_auto.py*

Estos serían los dos métodos creados para actualizar los datos de hora y fecha.

3.3.2. Obtención de datos de fecha y hora

Una vez que se hayan actualizado los registros con los datos de hora y fecha, se van a crear varios métodos que permitan acceder a dichos valores para conocer la hora y fecha actuales.

Lo primero que se va a hacer es elaborar métodos específicos para leer cada registro. Todos ellos harán una llamada al método elemental *leer* pasando como parámetro el registro específico, convertirán el dato recibido de binario con representación BCD a decimal y lo devolverán. Estos métodos son:

```
def leer_segundos(self):
    return bcd_to_int(self.leer(self.REG_SEGUNDOS))

def leer_minutos(self):
    return bcd_to_int(self.leer(self.REG_MINUTOS))

def leer_hora(self):
    return bcd_to_int(self.leer(self.REG_HORA)

def leer_dia_semana(self):
        return bcd_to_int(self.leer(self.REG_DIA_SEMANA))

def leer_dia_mes(self):
    return bcd_to_int(self.leer(self.REG_DIA))

def leer_mes(self):
    return bcd_to_int(self.leer(self.REG_MES))

def leer_anio(self):
    return bcd_to_int(self.leer(self.REG_ANIO))
```

Ahora, a partir de los métodos anteriores, se va elaborar uno que haga una llamada a todos ellos y, de este modo, devolver una tupla o array que contenga el valor de todos los parámetros en una variable.

Este método es el siguiente:

```
def leer_todo(self):
        return (self.leer_hora(), self.leer_minutos(), self.leer_segundos(),
                self.leer_dia_semana(), self.leer_dia_mes(), self.leer_mes(),
                self.leer_anio())
```

Para acabar, se ha realizado un script que muestra por pantalla los datos completos de hora y fecha.

El script, nombrado como “leer_datos” se muestra a continuación:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                              leer_datos.py                                 #
#                                                                            #
# Programa para visualizar los datos actuales de fecha y hora completos y    #
# temperatura.                                                               #
#                                                                            #
##############################################################################

import DS3231

RTC = DS3231.DS3231()

# Obtiene un vector con todos los datos
datos = RTC.leer_todo()

if datos[3] == 1:
    dia_semana = "Lunes"
elif datos[3] == 2:
    dia_semana = "Martes"
elif datos[3] == 3:
    dia_semana = "Miercoles"
elif datos[3] == 4:
    dia_semana = "Jueves"
elif datos[3] == 5:
    dia_semana = "Viernes"
elif datos[3] == 6:
    dia_semana = "Sabado"
elif datos[3] == 7:
    dia_semana = "Domingo"
else:
    raise ValueError("No se ha podido definir el dia de la semana")

# Calcula el año con 4 dígitos a partir del siglo definido y el dato año
anio = (DS3231.SIGLO - 1)*100 + datos[6]

# Muestra los valores con formato por pantalla
print("\n\tHora: %02d:%02d:%02d" % (datos[0], datos[1], datos[2]))
print("\n\tFecha: " + dia_semana + " %02d/%02d/%02d" %
      (datos[4], datos[5], anio))
```

El programa almacena en la variable *datos* todos los valores de cada uno de los parámetros devueltos por el método *leer_todo*. Las posiciones de estos parámetros en la tupla creada serían:

- datos[0]: hora (formato 24h).
- datos[1]: minutos.
- datos[2]: segundos.
- datos[3]: día de la semana. Valor numérico entre 1 y 7.
- datos[4]: día del mes.
- datos[5]: mes. Valor numérico entre 1 y 12.
- datos[6]: últimos dos dígitos del año.

En función del valor del elemento datos[3], se crea una variable de tipo string que contendrá el nombre del día de la semana actual para mostrarlo por pantalla más adelante.

Además, se calcula la variable *anio*, que será el año con cuatro cifras decimales ya que, en el registro únicamente se almacenan las dos últimas. Esto se realiza mediante el uso de la constante SIGLO que se mostró con anterioridad. Con la operación `anio = (DS3231.SIGLO - 1)*100 + datos[6]` se obtiene dicho valor de año.

Con todos ellos se va a mostrar el mensaje por pantalla dando formato a los datos.

Para ejecutar el script, desde el terminal se escribe:

```
sudo python leer_datos.py
```

Obteniendo como resultado:

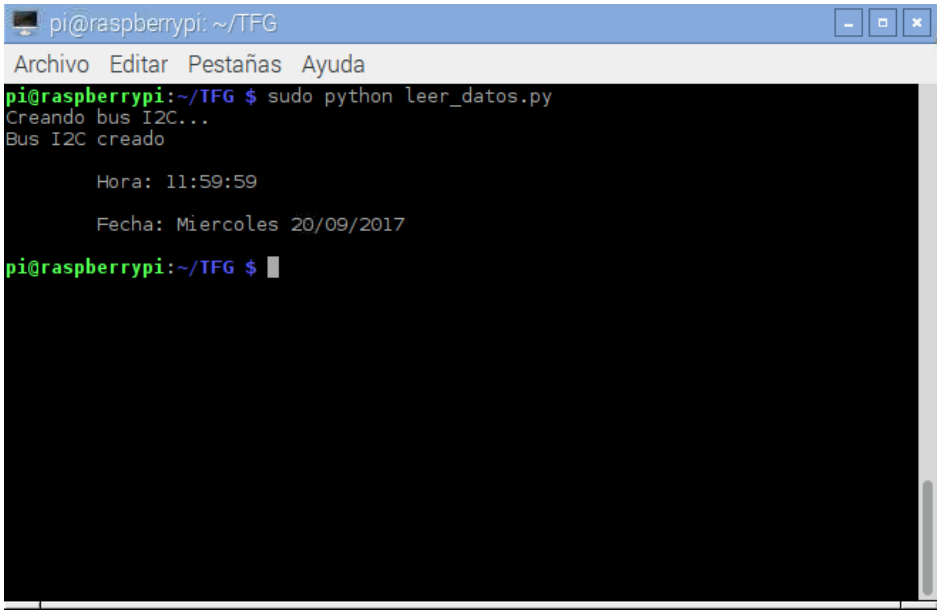

Ilustración 40. Ejecución del script *leer_datos.py*

De esta manera se pueden visualizar los datos completos y actualizados de hora y fecha y así, comprobar que la inicialización y el mantenimiento de los registros con valores actualizados se efectúan correctamente.

3.3.3. Temperatura

Como se mencionó en el apartado 2.4.2, al incorporar el DS3231 un sensor de temperatura para realizar calibraciones, es posible acceder a la lectura de valores de temperatura.

Mirando la Tabla 3 se observa que estas lecturas de temperatura se almacenan en los registros 0x11 y 0x12.

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

Ilustración 41. Registros de almacenamiento de temperatura

Observando la Ilustración 41 y la información proporcionada por el fabricante, la temperatura se representa con 10 bits y una resolución de 0.25 °C. La temperatura se almacena en los registros 11h y 12h con codificación binaria de complemento a 2. El byte del registro 0x11 (*Upper byte*) contiene la parte entera,

mientras que los bits 7 y 6 del registro 0x12 (*Lower byte*) contienen la parte fraccionaria.

Según esta representación, si se tuviera el número binario 0001100101, el valor decimal correspondiente sería +25.25 °C.

Con todo esto se va a crear un método que realice la lectura de estos dos registros y devuelva un valor decimal correspondiente a la lectura de temperatura del DS3231.

La definición de este método es la siguiente:

```
def obtener_temperatura(self):
        byte_msb = self.leer(self.REG_TEMP_MSB)
        byte_lsb = bin(self.leer(self.REG_TEMP_LSB))[2:].zfill(8)
        return byte_msb + int(byte_lsb[0])*2**(-1) + int(byte_lsb[1])*2**(-2)
```

El funcionamiento es el siguiente:

Primero se almacena en *byte_msb* todo el contenido del registro 0x11 y en la variable *byte_lsb* los dos primeros bits (7 y 6) del registro 0x12. Se realiza la conversión de estos dos bits a un número decimal multiplicando el bit 7 por 2^{-1} y el bit 6 por 2^{-2}. Este valor sumado a *byte_msb* será el que devuelva la función.

En este caso no es necesario realizar ninguna conversión ya que, como se ha dicho antes, el valor se almacena en binario con representación de complemento a 2.

Por último, se va a añadir al script *leer_datos.py* el uso de esta nueva función para que, además de mostrar la hora y fecha actualizadas, muestre también la temperatura.

Por lo que el nuevo script quedaría:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                              leer_datos.py                                 #
#                                                                            #
# Programa para visualizar los datos actuales de fecha y hora completos y    #
# temperatura.                                                               #
#                                                                            #
##############################################################################

import DS3231

RTC = DS3231.DS3231()

# Obtiene un vector con todos los datos
datos = RTC.leer_todo()
```

```
if datos[3] == 1:
    dia_semana = "Lunes"
elif datos[3] == 2:
    dia_semana = "Martes"
elif datos[3] == 3:
    dia_semana = "Miercoles"
elif datos[3] == 4:
    dia_semana = "Jueves"
elif datos[3] == 5:
    dia_semana = "Viernes"
elif datos[3] == 6:
    dia_semana = "Sabado"
elif datos[3] == 7:
    dia_semana = "Domingo"
else:
    raise ValueError("No se ha podido definir el dia de la semana")

# Calcula el año con 4 dígitos a partir del siglo definido y el dato año
anio = (DS3231.SIGLO - 1)*100 + datos[6]

# Muestra los valores con formato por pantalla
print("\n\tHora: %02d:%02d:%02d" % (datos[0], datos[1], datos[2]))
print("\n\tFecha: " + dia_semana + " %02d/%02d/%02d" %
      (datos[4], datos[5], anio))
print("\n\tTemperatura: " + str(RTC.obtener_temperatura()) + " *C\n")
```

Se realiza de nuevo la ejecución y se obtiene lo siguiente:

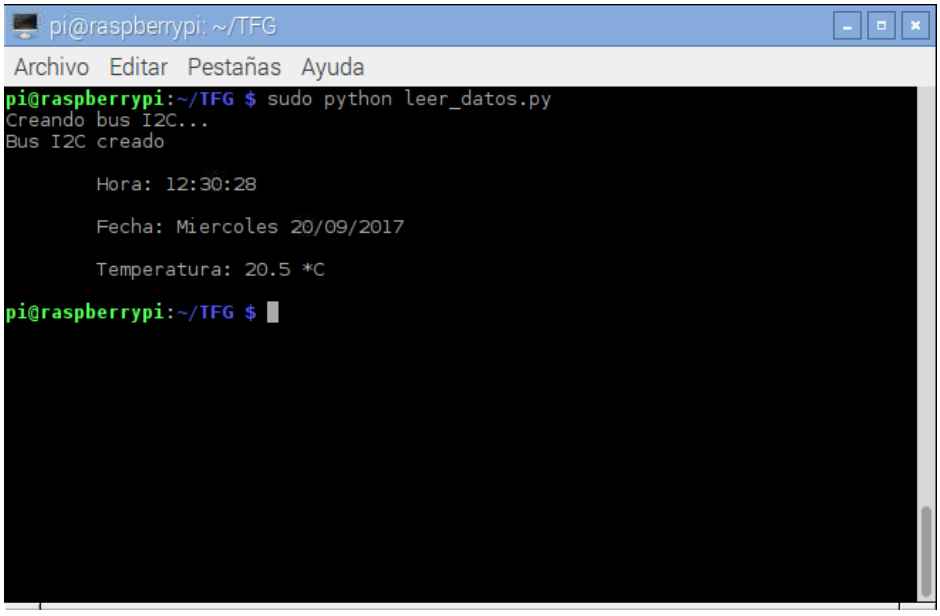

Ilustración 42. Ejecución del script *leer_datos.py* actualizado

3.3.4. Alarmas

Como ya se mencionó anteriormente, el DS3231 dispone de dos alarmas programables. La primera de ellas se configura escribiendo los registros del 07h al 0Ah, y la otra con los registros del 0Bh al 0Dh (Ver Tabla 3).

Si se estudian estos registros se puede observar que la configuración de ellos es muy similar a la de los registros donde se almacenan la hora y la fecha actualizadas. La diferencia está en que el día de la semana y el día del mes comparten un mismo registro (0Ah para la alarma 1 y 0Dh para la alarma 2), siendo seleccionable uno u otro mediante el bit 6 de los mismos. Si este bit vale 1, el dato almacenado será el día de la semana (*day*) mientras que, si vale 0, el dato será el día del mes (*date*).

Además, el conjunto de los bits 7 de cada uno de estos registros conforman la máscara de alarma y permiten configurar el tipo de alarma que se desea establecer.

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 20 Hour	10 Hour	Hour				Hours	1–12 + $\overline{AM}$/PM 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 20 Hour	10 Hour	Hour				Alarm 1 Hours	1–12 + $\overline{AM}$/PM 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 20 Hour	10 Hour	Hour				Alarm 2 Hours	1–12 + $\overline{AM}$/PM 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	—

Ilustración 43. Bits de la máscara de alarma

Los distintos modos de alarma que se enunciaron en el apartado 2.4.2 se obtienen mediante las distintas configuraciones que pueden adoptar estos bits de máscara. Todas ellas se muestran a continuación:

DY/$\overline{DT}$	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/$\overline{DT}$	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

Tabla 4. Configuración de los bits de máscara de alarma

A pesar de todas las configuraciones disponibles, aquí sólo se va a trabajar con los 3 modos que se describieron en el apartado 3.2, iguales para ambas alarmas.

La alarma 1 permite ajustar además los segundos a los que se desea activar, pero como no es del interés en este trabajo, cuando se configuren las alarmas, el valor de los segundos será 0.

Por lo tanto, los 3 modos con los que se trabajará y su máscara de bits correspondiente serán:

DY/$\overline{DT}$	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/$\overline{DT}$	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

Ilustración 44. Modos de alarma y su máscara de bits

Con todo esto se va a definir un método que reciba como parámetros el modo de alarma deseado, las horas, los minutos y un parámetro opcional que será el día de la semana o del mes en función del modo seleccionado.

Dicha función, nombrada “crear_alarma_1”, es:

```
def crear_alarma_1(self, modo, hora, minutos, dia=None):

	# MODO 1 - Alarma que se activa a la hora y minutos asignados
	#          (cualquier dia)
	if modo == 1:
		# Se definen los segundos a los que se activara la alarma (0)
		self.escribir(self.REG_A1_SEGUNDOS, 0x00)

		if hora < 0 or hora > 23:
			raise ValueError("Horas fuera de rango [0,23]")
		self.escribir(self.REG_A1_HORA, int_to_bcd(hora))

		if minutos < 0 or minutos > 59:
			raise ValueError("Minutos fuera de rango [0,59]")
		self.escribir(self.REG_A1_MINUTOS, int_to_bcd(minutos))

		# Se fija para que la alarma solo se active a la hora, minutos
		# y segundos especificados
		self.escribir(self.REG_A1_DIA, 0x80)

	# MODO 2 - Alarma que se activa a la hora, minutos y dia del mes
	#          asignados
	elif modo == 2:
		# Se definen los segundos a los que se activara la alarma (0)
		self.escribir(self.REG_A1_SEGUNDOS, 0x00)

		if hora < 0 or hora > 23:
			raise ValueError("Horas fuera de rango [0,23]")
		self.escribir(self.REG_A1_HORA, int_to_bcd(hora))

		if minutos < 0 or minutos > 59:
			raise ValueError("Minutos fuera de rango [0,59]")
		self.escribir(self.REG_A1_MINUTOS, int_to_bcd(minutos))

		if dia_mes < 1 or dia_mes > 31:
			raise ValueError("Dia del mes fuera de rango [1,31]")
		self.escribir(self.REG_A1_DIA, int_to_bcd(dia_mes))

	# MODO 3 - Alarma que se activa a la hora, minutos y dia de la semana
	#          asignados
	elif modo == 3:
		# Se definen los segundos a los que se activara la alarma (0)
		self.escribir(self.REG_A1_SEGUNDOS, 0x00)

		if hora < 0 or hora > 23:
			raise ValueError("Horas fuera de rango [0,23]")
		self.escribir(self.REG_A1_HORA, int_to_bcd(hora))

		if minutos < 0 or minutos > 59:
			raise ValueError("Minutos fuera de rango [0,59]")
		self.escribir(self.REG_A1_MINUTOS, int_to_bcd(minutos))

		if dia_semana < 1 or dia_semana > 7:
			raise ValueError("Dia del mes fuera de rango [1,7]")
		self.escribir(self.REG_A1_DIA,int_to_bcd(dia_semana)+0b1000000)
		# Bit 6 a 1 para habilitar dia de la
		# semana en vez de dia del mes

	# ERROR
	else:
		raise ValueError("Modo de alarma seleccionado erroneo")
```

La función equivalente para la alarma 2 será similar. La diferencia está en que la dirección de los registros en los que hay que escribir es distinta y que no hace falta definir los segundos a 0, ya que la segunda alarma no abarca su configuración (Ver Ilustración 44).

Una vez programada la alarma, hay que activarla para permitir que se provoque una interrupción por el pin $\overline{INT}/SQW$. Para ello es necesario analizar el registro de control (0Eh).

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

Ilustración 45. Registro de control

Los bits del registro de control que van a ser de interés son los 3 menos significativos: *INTCN*, *A2IE* y *A1IE*.

El bit *INTCN* controla la señal del pin $\overline{INT}/SQW$. Cuando el valor de este pin es 0, la salida del pin es una onda cuadrada cuya frecuencia se controla modificando los valores de los bits *RS2* y *RS1*, cosa que no es de interés en este trabajo. Cuando el valor de este bit es 1, queda habilitada la salida de interrupción de este pin. La interrupción se produce cuando los valores de los registros que almacenan los datos de hora y fecha actuales coinciden con los programados en cada una de las alarmas; siempre y cuando las alarmas estén activadas.

Para activar/desactivar cada una de las alarmas es necesario modificar los dos últimos bits del registro: bit 1 para la alarma 2 y el bit 0 para la alarma 1.

Cuando el valor del bit es 0 la alarma está desactivada; por el contrario, si el bit es 1 la alarma correspondiente a dicho bit (1 ó 0) estará activada y, por lo tanto, permitirá la interrupción por el pin $\overline{INT}/SQW$ cuando se alcance la hora y fecha programadas (siempre que el bit *INTCN* del registro esté a 1 como se comentó antes).

Con esto se van a realizar los métodos correspondientes a cada alarma que permitan activar y desactivar estas.

A continuación se muestran los métodos para activar y desactivar la alarma 1. Los referentes a la alarma 2 son análogos y se pueden ver en el módulo completo.

```
def activar_alarma_1(self):
        # Comprueba si ya está activa la alarma (bit 0 a 1)
        if (self.leer(self.REG_CONTROL) & 0b1):
                # Ya está activa, no hace nada
                pass
        else:
                # No está activa, escribe bit 0 a 1
                estado_registro = self.leer(self.REG_CONTROL)
                self.escribir(self.REG_CONTROL, estado_registro + 0b1)

def desactivar_alarma_1(self):
        # Comprueba si ya está desactivada la alarma (bit 0 a 0)
        if ((self.leer(self.REG_CONTROL) & 0b1) == False):
                # Ya está desactiva, no hace nada
                pass
        else:
                # No está desactivada, escribe bit 0 a 0 (restándole 1)
                estado_registro = self.leer(self.REG_CONTROL)
                self.escribir(self.REG_CONTROL, estado_registro - 0b1)
```

Como se puede observar, antes de cambiar el valor del bit se comprueba su estado actual. Si el bit tiene el valor deseado, no se modifica; si no, se cambia el valor para activar o desactivar la alarma correspondiente.

Antes de crear un script de prueba para comprobar el funcionamiento de las alarmas y su interrupción se va a realizar el estudio de otro registro relacionado con las alarmas: el registro de estado (0Fh).

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

Ilustración 46. Registro de estado

En este registro, los bits que serán de interés son los dos menos significativos: *A2F* y *A1F*. Estos dos bits se corresponden con el *flag* de la alarma 2 y el *flag* de la alarma 1, respectivamente.

Estos *flags* indican la alarma que ha sido activada. Si el bit *A2F* vale 1, quiere decir que se han alcanzado los valores de fecha y hora almacenados en los registros de la alarma 2. De manera análoga, sucede lo mismo con el bit *A1F* y la alarma 1.

Estos bits únicamente pueden ser cambiados a 0, si se intentan cambiar a 1 se quedarán como están.

Para que una alarma se active, es necesario que los *flags* estén a 0; si no, no se producirá la interrupción. Tras realizar pruebas se ha comprobado que tienen que estar ambos bits a 0 para activar la interrupción de cualquiera de las dos alarmas, ya que, como cabría esperar, el *flag* de la alarma 2 es independiente de la alarma 1; pero no es así, es necesario que ambos *flags* estén a 0.

Con todo esto, el último método referente a las alarmas, encargado de cambiar los flags a 0 será:

```
def borrar_flags(self):
       # Flag alarma 1
       if (self.leer(self.REG_STATUS) & 0b1):
              estado_registro = self.leer(self.REG_STATUS)
              self.escribir(self.REG_STATUS, estado_registro - 0b1)

       #Flag alarma 2
       if (self.leer(self.REG_STATUS) & 0b10):
              estado_registro = self.leer(self.REG_STATUS)
              self.escribir(self.REG_STATUS, estado_registro - 0b10)
```

Para acabar, se ha creado un método más que permita leer los registros de cada alarma y, así, conocer los datos de esta; es decir, saber en qué modo y a qué hora, minutos y día está programada (en caso de que lo esté).

El método para la alarma 1 se muestra a continuación. Para la alarma 2 sería análogo.

```
def leer_alarma_1(self):
       if (self.leer(self.REG_A1_DIA) & 0x80):
              modo = 1
              dia = None
       elif ((self.leer(self.REG_A1_DIA) & 0x40) == False):
              modo = 2
              dia = bcd_to_int(self.leer(self.REG_A1_DIA))
       elif (self.leer(self.REG_A1_DIA) & 0x40):
              modo = 3
              dia = bcd_to_int(self.leer(self.REG_A1_DIA) - 0x40)
       else:
              raise ValueError("No se ha podido realizar lectura de " +
                                          "la ALARMA 1")

       return (modo, bcd_to_int(self.leer(self.REG_A1_HORA)),
                     bcd_to_int(self.leer(self.REG_A1_MINUTOS)), dia)
```

Ahora ya se va a realizar un script que permita comprobar el funcionamiento de las alarmas. Este script permitirá la introducción manual de los datos para cada alarma, la creará y la activará, mostrando un mensaje por pantalla indicando los datos de programación para comprobar el funcionamiento del método de lectura de alarmas (leer_alarma_1 y leer_alarma_2). Después, el programa no realizará ninguna acción más a la espera de la interrupción provocada por el RTC. Cuando la interrupción se provoque mostrará un mensaje por el terminal indicando a la hora a la que se produjo la interrupción.

El script, nombrado como *prueba_alarmas.py*, queda:

```
#!/usr/bin/python3
#-*- coding: iso-8859-15 -*-

##############################################################################
#                                                                            #
#                            prueba_alarmas.py                               #
#                                                                            #
# Programa para configurar las dos alarmas y recibir su interrupción con     #
# un mensaje por pantalla.                                                   #
# Al salir del programa (Ctrl+C) se limpian los GPIO, se desactivan las      #
# alarmas en caso de que no se haya recibido y se borra el flag en caso de   #
# que sí se haya recibido.                                                   #
#                                                                            #
##############################################################################

# Importación de las librerías
import RPi.GPIO as GPIO
import DS3231

RTC = DS3231.DS3231()

# Se configuran los pines en modo BCM (Broadcom)
GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
# Configuración del pin 17 como entrada con resistencia pull-up
GPIO.setup(17, GPIO.IN, pull_up_down=GPIO.PUD_UP)

# Rutina de tratamiento de la interrupción
def interrupcion(channel):
    datos = RTC.leer_todo()
    print("\n\tALARMA RECIBIDA A LAS: " +
          "%02d:%02d:%02d" % (datos[0], datos[1], datos[2]))
    RTC.borrar_flags()

# Declaración de la interrupción:
# Pin, tipo de interrupción y rutina de tratamiento asociada
GPIO.add_event_detect(17, GPIO.FALLING, callback=interrupcion,
bouncetime=300)

# Declaración de la alarma
print("\n\tPROGRAMACION DE ALARMA 1")
modo_1 = input("\nIntroduzca modo de alarma: ")
hora_1 = input("Introduzca hora (formato 24h): ")
minutos_1 = input("Introduzca minutos (0-59): ")
if modo_1 == 1:
    dia_1 = None
elif modo_1 == 2:
    dia_1 = input("Introduzca dia del mes: ")
elif modo_1 == 3:
    dia_1 = input("Introduzca dia de la semana (1-7): ")
else:
    raise ValueError ("El modo seleccionado no es correcto")

print("\n\tPROGRAMACION DE ALARMA 2")
modo_2 = input("\nIntroduzca modo de alarma: ")
hora_2 = input("Introduzca hora (formato 24h): ")
minutos_2 = input("Introduzca minutos (0-59): ")
if modo_2 == 1:
    dia_2 = None
elif modo_2 == 2:
    dia_2 = input("Introduzca dia del mes: ")
elif modo_2 == 3:
    dia_2 = input("Introduzca dia de la semana (1-7): ")
else:
    raise ValueError ("El modo seleccionado no es correcto")
```

```
# Se crea y activan las alarmas y se muestra mensaje de confirmación por
# pantalla para comprobar la lectura de las mismas
RTC.crear_alarma_1(modo_1, hora_1, minutos_1, dia_1)
RTC.activar_alarma_1()
alarma_1 = RTC.leer_alarma_1()
if alarma_1[3] == None:
    print("\nAlarma 1 configurada en modo %d a las %02d:%02d"
          % (alarma_1[0], alarma_1[1], alarma_1[2]))
else:
    print("\nAlarma 1 configurada en modo %d a las %02d:%02d del dia %02d"
          % (alarma_1[0], alarma_1[1], alarma_1[2], alarma_1[3]))

RTC.crear_alarma_2(modo_2, hora_2, minutos_2, dia_2)
RTC.activar_alarma_2()
alarma_2 = RTC.leer_alarma_2()
if alarma_2[3] == None:
    print("\nAlarma 2 configurada en modo %d a las %02d:%02d"
          % (alarma_2[0], alarma_2[1], alarma_2[2]))
else:
    print("\nAlarma 2 configurada en modo %d a las %02d:%02d del dia %02d"
          % (alarma_2[0], alarma_2[1], alarma_2[2], alarma_2[3]))

# Ejecución del programa
try:
    # Espera a recibir la interrupción
    while(True):
        pass

# El programa se ejecuta hasta que se manda la señal de finalización
# desde el teclado
except KeyboardInterrupt:
    GPIO.cleanup()
    RTC.desactivar_alarma_1()
    RTC.desactivar_alarma_2()
    RTC.borrar_flags()
```

De manera análoga a los scripts de prueba realizados anteriormente, para iniciar la ejecución de este habrá que introducir la siguiente línea de comando en el terminal (estando en el directorio en el que se encuentra el fichero .py):

```
sudo python prueba_alarmas.py
```

A continuación se muestran algunas imágenes de la ejecución del mismo:

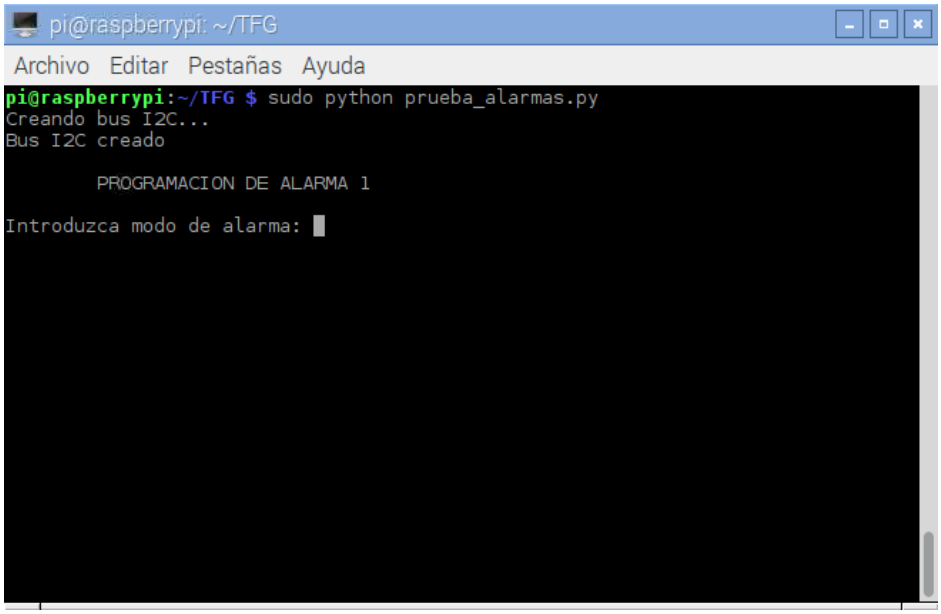

Ilustración 47. Ejecución del script *prueba_alarmas.py*

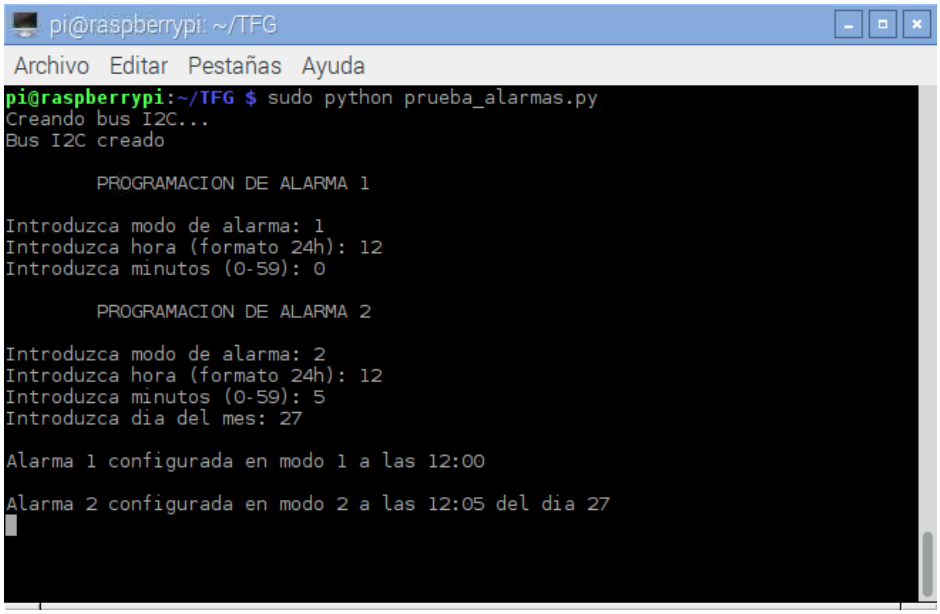

Ilustración 48. Introducción de datos solicitados

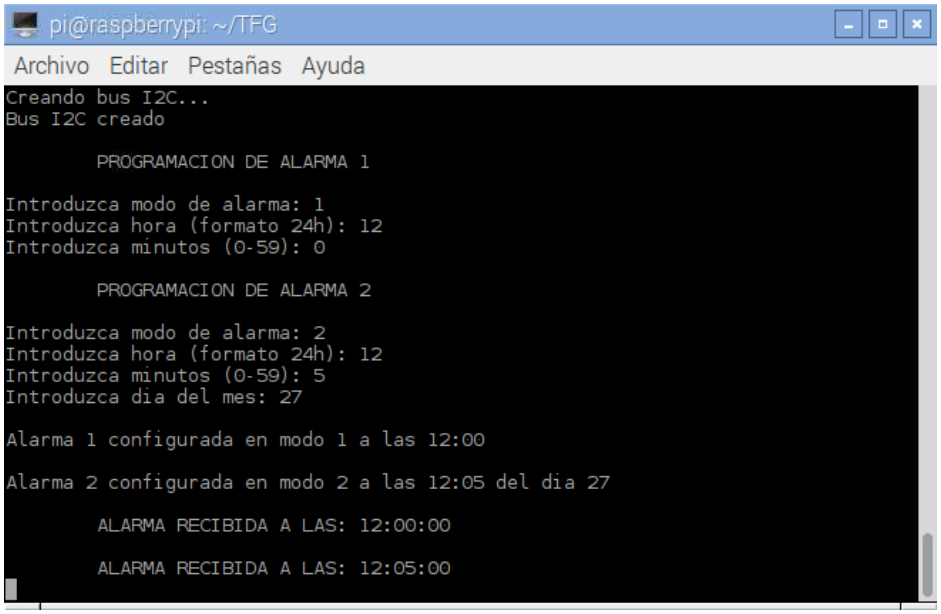

Ilustración 49. Mensajes de alarma recibida

4. APLICACIONES PRÁCTICAS

A continuación se van a presentar dos casos de aplicaciones del módulo creado para ver con más claridad su finalidad.

El primero de ellos es una subrutina que se basa en una alarma que se activa al inicio de la ejecución de esta y, de manera periódica, cada hora a partir de ese momento.

Cada vez que se reciba la alarma se llevará a cabo el proceso de tratamiento de la interrupción, el cual consiste en realizar la lectura de un sensor de temperatura y humedad y almacenar la medición en un fichero de texto, indicando la hora y la fecha de dicha lectura.

El sensor empleado para esta aplicación es el DHT22. El esquema de montaje se muestra a continuación:

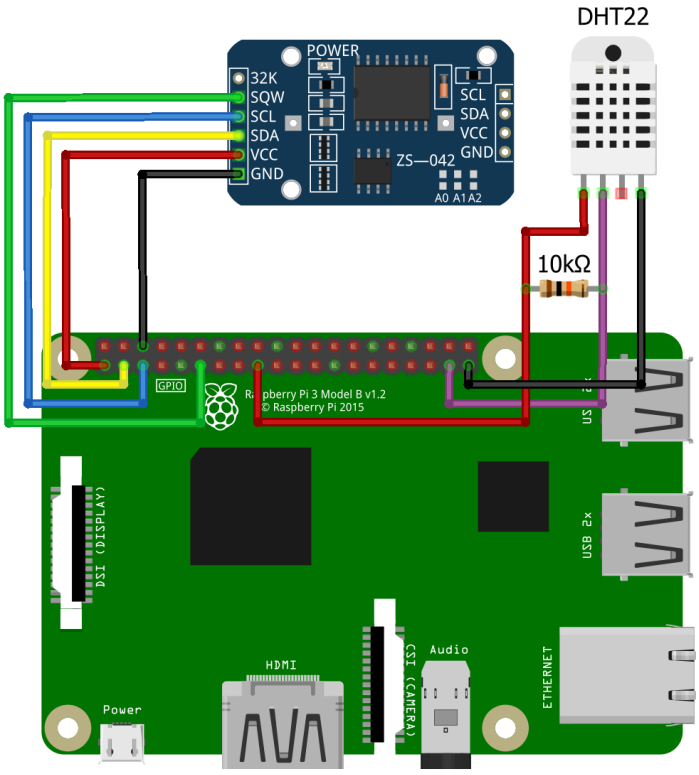

Ilustración 50. Esquema de montaje aplicación 1

El código (debidamente comentado para su comprensión) se adjunta en el ANEXO, junto con el fichero de texto generado en una ejecución de prueba del mismo (*registro_ejecucion.txt*).

La otra aplicación es un control de riego. Permite al usuario programar dos alarmas al día para activar el riego, además de poder configurar la duración de este.

La aplicación constará de un menú que se muestra a través del terminal con diversas opciones.

Ilustración 51. Menú de usuario del control de riego

Para simular el riego se emplea un diodo LED conectado a uno del GPIO de la Raspberry según muestra el esquema de la Ilustración 52. El LED se enciende para indicar que se activaría el riego.

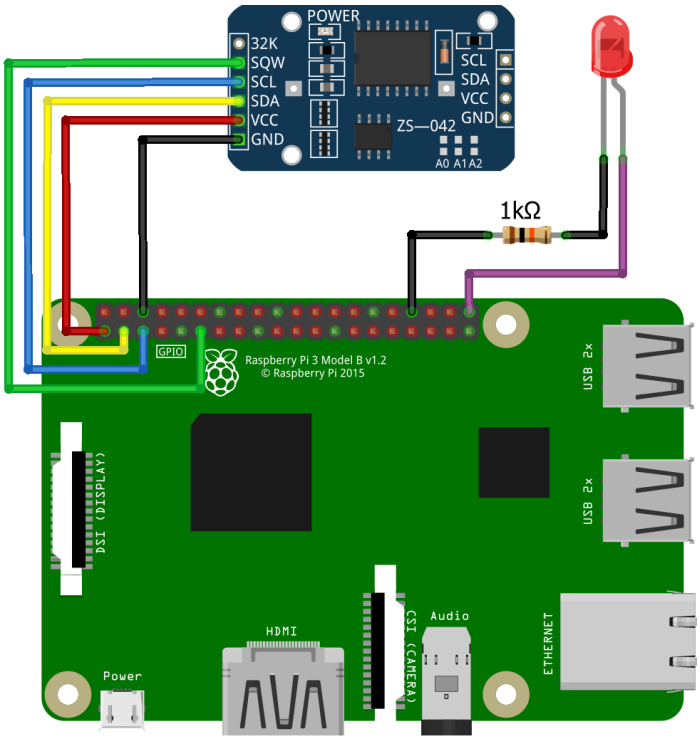

Ilustración 52. Esquema de montaje aplicación 2

Para comprobar el funcionamiento del programa, todas las acciones que se realizan en él se almacenan en un fichero de texto, indicando la hora y fecha exactas a la que se produjeron.

El código del programa (*control_riego.py*) se adjunta en el ANEXO.

5. ESTUDIO ECONÓMICO

A continuación, se va a realizar un presupuesto de lo que ha supuesto el diseño y la elaboración del módulo presentado anteriormente.

Este presupuesto es aproximado, ya que únicamente se van a tener en cuenta los costes directos. Esto se debe a que el enfoque inicial de este proyecto es meramente académico y de aprendizaje. No obstante con esto se realiza una aproximación de lo que supondría la adición del presente proyecto a sistemas que se encuentran actualmente en el mercado.

La siguiente tabla recoge los costes de todos los materiales empleados en el proyecto.

Descripción del elemento	Modelo	Cantidad	Precio (€/Ud.)	Subtotal (€)
Raspberry Pi	3B	1	30,19	30,19
Reloj de tiempo real	DS3231	1	3,54	3,54
Batería de litio	LIR2032	1	1,4	1,4
			SUMA:	35,13 €
			I.V.A. (21%):	7,38 €
			TOTAL:	**42,51 €**

Tabla 5. Costes materiales

Lo siguiente será realizar la estimación del tiempo empleado en el desarrollo del proyecto, incluyendo las horas de aprendizaje previas para dicho cometido.

Descripción	Horas	Precio (€/h)	Subtotal (€)
Aprendizaje de Python	40	7,88	315,2
Aprendizaje de uso de Raspberry Pi	15	7,88	118,2
Configuración inicial de Raspberry Pi	5	7,88	39,4
Tiempo estimado de elaboración del módulo	60	7,88	472,8
		SUMA:	945,60 €
		I.V.A. (21%):	198,58 €
		TOTAL:	**1.144,18 €**

Tabla 6. Costes de personal

El precio por hora de trabajo ha sido obtenido a partir de las tablas salariales publicadas en el Boletín Oficial del Estado, en el que se establece que el salario base bruto al año es de 17404 € para Diplomados y titulados 1er ciclo universitario. (Resolución de 9 de octubre de 2013, de la Dirección de Empleo, por la que se registra y publica el XVII Convenio colectivo nacional de empresas de ingeniería y oficinas de estudios técnicos).

Con todo ello ya se puede estimar el total de los costes directos.

Descripción	Subtotal (€)
Costes materiales	35,13
Costes de personal	945,6
SUMA:	980,73 €
I.V.A. (21%):	205,95 €
TOTAL:	**1.186,68 €**

Tabla 7. Costes directos totales

6. CONCLUSIONES

Tras la realización del presente trabajo se ha podido realizar un análisis de los factores que han intervenido en él.

Por una parte, el lenguaje de programación: Python. Como ya se mencionó al comienzo de este documento, ha resultado ser un lenguaje muy sencillo y fácil de aprender; además de la comodidad que supone su uso. Su versatilidad para aplicaciones del estilo a la tratada en este trabajo lo hace, como se ha podido comprobar, de los lenguajes de programación más usados en la actualidad.

Por otra parte, la encargada de llevar todo a cabo: la Raspberry Pi. Destaca el potencial del que dispone este ordenador de bolsillo, a pesar de su reducido tamaño y consumo.

Por último, en referencia a lo expuesto en el presupuesto, más concretamente en la tabla de costes materiales, se observa el bajo coste que suponen los materiales necesarios para la implementación de este sistema. Además, destacar que tanto el software que se emplea y sus actualizaciones son completamente gratuitas.

6.1. Líneas futuras

El enfoque dado en las aplicaciones prácticas mostradas en el apartado 4 está bastante relacionado con los sistemas domóticos que se pueden implementar con Raspberry Pi, cuando es esta la que actúa como servidor. Es en esas aplicaciones en las que mejor se observa la implementación de todo lo anterior.

En esos casos lo desarrollado en este proyecto cobraría gran importancia y utilidad debido a la carga de trabajo de la que se liberaría al procesador. A pesar de que la Raspberry Pi estaría conectada a la red y dispondría de datos de fecha y hora actualizados, si se programan los eventos como alarmas del DS3231 se evitaría que el procesador estuviera continuamente realizando lecturas de fecha y hora para realizar las acciones oportunas.

Si se quisiese ir un poco más allá, se podrían emplear varios módulos RTC para controlar eventos distintos. Bastaría con conectarlos al bus I2C, configurando direcciones distintas para cada uno de ellos, y conectar los pines de alarma de todos ellos a los GPIO de la Raspberry. Después se programarían tratamientos de interrupción distintos para cada uno de estos. Todo ello sin incrementar apenas los costes, como ya se ha mostrado con anterioridad.

7. BIBLIOGRAFÍA

- García, Vicente (2012). Introducción al I2C BUS. Fecha de consulta: septiembre de 2017. Recuperado de https://www.diarioelectronicohoy.com/blog/introduccion-al-i2c-bus

- J. Carletti, Eduardo (2007). Comunicación – Bus I2C. Descripción y funcionamiento por Eduardo J. Carletti. Fecha de consulta: septiembre de 2017. Recuperado de http://robots-argentina.com.ar/Comunicacion_busI2C.htm

- Downey, A. (2015), *Think Python. How to Think Like a Computer Scientist*, Needham, Massachusetts: Green Tea Press.

- Torres, Héctor (2014). Python: I2C, uso y configuración. Fecha de consulta: septiembre de 2017. Recuperado de https://hetpro-store.com/TUTORIALES/python-i2c-uso-y-configuracion/

- Adam. Raspberry Pi Resources. Using the I2C Interface. Fecha de consulta: septiembre de 2017. Recuperado de http://www.raspberry-projects.com/pi/programming-in-python/i2c-programming-in-python/using-the-i2c-interface-2

- Raspberry Pi 3 Model B Specifications. Fecha de consulta: junio de 2017. Recuperado de https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

- Guía Raspberry Pi para principiantes. Fecha de consulta: junio de 2017. Recuperado de https://www.raspberryshop.es/guia-completa-raspberry-pi.php

- Croston, Ben. Raspberry-gpio-python. RPi.GPIO module basics. Fecha de consulta: julio de 2017. Recuperado de https://sourceforge.net/p/raspberry-gpio-python/wiki/BasicUsage/

- Croston, Ben. Raspberry-gpio-python. Inputs. Fecha de consulta: julio de 2017. Recuperado de https://sourceforge.net/p/raspberry-gpio-python/wiki/Inputs/

- AntMan232. Raspberry Pi I2C (Python). Fecha de consulta: septiembre de 2017. Recuperado de http://www.instructables.com/id/Raspberry-Pi-I2C-Python/

- Maxim Integrated. DS1307 datasheet. Maxim Integrated Products, Inc. Sunnyvale, CA. 2015.

- Maxim Integrated. DS1307 datasheet. Maxim Integrated Products, Inc. Sunnyvale, CA. 2015.

- Llamas, Luis (2016). Reloj y calendario en Arduino con los RTC DS1307 y DS3231. Fecha de consulta: octubre de 2017. Recuperado de http://www.instructables.com/id/Raspberry-Pi-I2C-Python/

- Garba, Adam (2016). Raspberry Pi 2 IoT: Thingspeak & DHT22 Sensor. Fecha de consulta: noviembre de 2017. Recuperado de https://www.hackster.io/adamgarbo/raspberry-pi-2-iot-thingspeak-dht22-sensor-b208f4

- Resolución de 9 de octubre de 2013, de la Dirección de Empleo, por la que se registra y publica el XVII Convenio colectivo nacional de empresas de ingeniería y oficinas de estudios técnicos. Boletín Oficial del Estado, núm. 256, de 25 de octubre de 2013, páginas 86811 a 86838. https://boe.es/diario_boe/txt.php?id=BOE-A-2013-11199