
# MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

# DESARROLLO DE UNA MÁQUINA AUTO VENDING

Autor: Manchado González, Felipe

Director: García de Viguera Cabrera, Francisco

Madrid

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

DESARROLLO DE UNA MÁQUINA AUTO VENDING

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2022/2023 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Felipe Manchado González. Fecha: 21 / 06 / 2023

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO


Fdo.: García de Viguera Cabrera, Francisco. Fecha: 21 / 06 / 2023


# MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

## DESARROLLO DE UNA MÁQUINA AUTO VENDING

Autor: Manchado González, Felipe

Director: García de Viguera Cabrera, Francisco

Madrid

# DESARROLLO DE UNA MÁQUINA DE AUTOVENDING

**Autor: Felipe Manchado González**
Director: García de Viguera Cabrera, Francisco
Entidad Colaboradora: Know Ice S.L.

## RESUMEN DEL PROYECTO

El objeto de este proyecto es el diseño y desarrollo de una máquina destinada a la venta no supervisada de hielo alimenticio. La empresa colaboradora del proyecto, Know Ice S.L., basa la mayor parte de sus actividades en la producción y distribución de hielo destinado al consumo particular.

Mediante la producción de un dispositivo de venta capaz de operar de forma desatendida, Know Ice S.L. pretende extender su mercado de venta mediante una red de equipos frigoríficos capaz de realizar el proceso de almacenamiento y venta de hielo.

Mediante este proyecto se estudia la viabilidad de dicho dispositivo, a través del prototipado de un modelo que cuente con las características básicas para dar viabilidad a un equipo apto para la distribución, cumpliendo con las bases de calidad exigibles por la Unión Europea.

Adicionalmente a estas cualidades, el equipo de venta debe contar con las funcionalidades necesarias para la supervisión a distancia del contenido del mismo. De tal forma que un operario sea capaz de monitorizar los quipos puestos en servicio para la reposición de los bienes de venta.

**Palabras clave:** Plataforma de pago, terminal de pago, células de peso, API, amplificadores diferenciales, micro Python, Desarrollo Back End, comunicación HTTP, Raspberry Pi Pico

### 1. Introducción

El presente proyecto consiste en el diseño y fabricación de un dispositivo de auto vending destinado a la distribución y venta de hielo alimentario.

El objetivo del proyecto por lo tanto consiste en el desarrollo de las técnicas y programas de control de una máquina de funcionamiento autónomo para la venta al público.

### 2. Definición del proyecto

La primera parte del proyecto consiste en el desarrollo de una Interfaz de Programación de Aplicaciones (API) para la simulación de un entorno web en el que se ejecute una estructura Back End, esto es en esencia el tipo de arquitectura web encargada del manejo

de operaciones y solicitudes por los clientes. La estructura Back End se complementa con el desarrollo de una plataforma Front End, la cual realiza la función de interfaz Hombre-Máquina (refiriéndose el término “máquina” al apartado web Back End descrito con anterioridad). Para el desarrollo Back End del presente proyecto se empleará el entorno de desarrollo Node.js

El entorno de simulación Back End desarrollado para este proyecto hace a su vez la función de plataforma de Base de Datos. Sin embargo, de cara al futuro del proyecto, se deberá considerar complementar el entorno Back End finalmente integrado con un sistema de *DataBase* basado en SQL para mayor seguridad, fiabilidad y robustez del sistema.

Para realizar las funciones de control del dispositivo de venta, hará uso del microcontrolador Raspberry Pi Pico W. Este emplea un lenguaje de programación simplificado basado en Python para su control. Como plataforma de desarrollo y comunicación con el microcontrolador se hará uso de la aplicación Thonny, la cual permite la conectividad y desarrollo de lenguaje MicroPython para microcontroladores.

La placa microcontroladora centralizará las funciones de control, mientras que paralelamente mantendrá una comunicación mediante protocolo web con la plataforma de pago instalada en la máquina.

Para realizar las funciones de lectura de tarjeta y transacción bancaria, se inició una relación cooperativa con la empresa Paytef. Las actividades de esta empresa se centran en el desarrollo de plataformas de pago. Al mismo tiempo, Paytef seencuentra asociada a PAX, empresa desarrolladora de terminales de pago y datáfonos. Entre los productos de PAX, Paytef ofrece una gama de plataformas de pago orientadas al autovending.

Tras una serie de negociaciones, la empresa Paytef se ofreció a poner a nuestra disposición un servicio de simulación de pagos y un terminal de pago destinado al desarrollo de prototipos, modelo A50.

Esto permite el desarrollo de un código compatible con el terminal de pago, similar al programa definitivo que se instalará en el producto final, en donde se hará uso del teminal de pago modelo IM30, orientado al auto vending.

Para la medida del hielo sustraído y el posterior cálculo del coste de la transacción realizada, se ha realizado un diseño en torno al uso de células de carga de precisión en la base de la cámara frigorífica, haciendo la función de puntos de apoyo de la propia máquina.

Estas células de peso de alta precisión han sido provistas por la distirbuidora Balanzas Galicia S.L., cuyas actividades se centran en la fabricación de sistemas de medición de alta calidad con aplicabilidad a entornos de venta basados en el peso.

Por último, el proyecto se centra en el desarrollo de un dispositivo capaz de facilitar la instalación de los equipos.

Este dispositivo se debe a la importancia de la correcta calibración de los puntos de apoyo del equipo de venta.

Durante la instalación de la cámara frigorífica, resulta imprescindible que los puntos de apoyo de esta, conformados por las células de carga, soporten un peso similar entre ellas.

De haber un desequilibrio podría significar que la máquina de auto vending se encuentre torcida, sobrecargando una o varias de las células de peso , pudiendo deteriorar la medida o potencialmente dañar a la larga dicha célula.

## 3. Resultados

A lo largo del proyecto se realizará el desarrollo de una plataforma Back End de ejecución asíncrona, capaz de realizar funciones de consulta y modificación de datos.

Esta plataforma será capaz de reunir la información generada por los distintos dispositivos de venta puestos en servicio, siendo capaz de almacenar información general sobre estos y sus actividades.

Esta plataforma también cuenta con la función clave de actualizar la información almacenada en los dispositivos de control, tal como la de precio de venta de producto, número de serie de Know Ice S.L. y su ubicación. El dispositivo de control por su parte empleará esta información para comunicar adecuadamente su estado y realizar sus funciones.

Otro de los resultados obtenidos consiste en la construcción de funciones capacitadas para realizar las complejas tareas de control del dispositivo instalado en la máquina de venta. Estas funciones deben realizar la actualización de información y comunicación con la plataforma web, comunicación con la plataforma de pago y control de los actuadores distribuidos por el dispositivo.

Por último, se diseña y se completa el prototipo de nivelador de células, destinado al equilibrado de las células de carga del equipo de venta.

Este dispositivo, alimentado desde un puerto USB a través de la Raspberry Pi Pico, se conecta a las células de carga y, mediante un esquema de amplificación diferencial, mide la porción de peso soportada por cada punto de apoyo.

De esta forma, a través de una pantalla LCD instalada en el propio dispositivo, el operario encargado de la instalación de la máquina frigorífica será capaz de emplear la información ofrecida por este para el balanceo de cargas.

# DEVELOPMENT OF AN AUTOVENDING MACHINE

**Author: Felipe Manchado González**
Director: García de Viguera Cabrera, Francisco
Collaborating Entity: Know Ice S.L.

## PROJECT SUMMARY

The purpose of this project is the design and development of a machine for the unsupervised sale of food ice. The project partner company, Know Ice S.L., bases most of its activities on the production and distribution of ice for consumption.

Through the production of a sales device capable of operating unattended, Know Ice S.L. intends to expand its sales market through a network of refrigerated equipment capable of carrying out the ice storage and sales process.

Through this project, the viability of said device is studied, through the prototyping of a device that has the basic characteristics to give viability to a suitable equipment for distribution, complying with the quality bases required by the European Union.
Adicionalmente a estas cualidades, el equipo de venta debe contar con las funcionalidades necesarias para la supervisión a distancia del contenido del mismo. De tal forma que un operario sea capaz de monitorizar los quipos puestos en servicio para la reposición de los bienes de venta.

**Key words:** Payment platform, payment terminal, weight cells, API, differential amplifiers, micro Python, Back End Development, HTTP communication, Raspberry Pi Pico

### 1. Introduction

This project consists of the design and manufacture of an auto vending device for the distribution and sale of food ice.

The objective of the project therefore consists in the development of control techniques and programs for an autonomous operation machine for sale to the public.

### 2. Definition of the Project

The first part of the project consists of the development of an Application Programming Interface (API) for the simulation of a web environment in which a Back End structure is executed, this is essentially the type of web architecture in charge of managing operations and customer requests. The Back End structure is complemented by the development of a Front End platform, which performs the Human-Machine interface function (the term "machine" referring to the Back End web section described above). For the Back End

development of this project, the Node.js development environment will be used.El entorno de simulación Back End desarrollado para este proyecto hace a su vez la función de plataforma de Base de Datos, sin embargo, de cara al futuro del proyecto, se deberá considerar complementar el entorno Back End finalmente integrado con un sistema de *DataBase* basado en SQL para mayor seguridad, fiabilidad y robustez del sistema.

To perform the control functions of the vending device, it will use the Raspberry Pi Pico W microcontroller. It uses a simplified programming language based on Python for its control. As a platform for development and communication with the microcontroller, the Thonny application will be used, which allows connectivity and development of the MicroPython language for microcontrollers.

The microcontroller board will centralize the control functions, while at the same time maintaining communication via web protocol with the payment platform installed in the machine.

To carry out the card reading and bank transaction functions, a cooperative relationship was started with the company Paytef. The activities of this company are focused on the development of payment platforms. At the same time, Paytef is associated with PAX, a company that develops payment terminals and dataphones. Among the PAX products, Paytef offers a range of payment platforms oriented towards autovending.

After a series of negotiations, the Paytef company offered to make available to us a payment simulation service and a payment terminal for the development of prototypes, model A50.

This allows the development of a code compatible with the payment terminal, similar to the final program that will be installed in the final product, where the IM30 model payment terminal will be used, oriented to auto vending.

For the measurement of the stolen ice and the subsequent calculation of the cost of the transaction carried out, a design has been carried out around the use of precision load cells at the base of the cold room, acting as support points for the machine itself. .

These high-precision weight cells have been provided by the distributor Balanzas Galicia, whose activities are focused on manufacturing high-quality measurement systems, applicable to weight-based sales environments.

Finally, the project focuses on the development of a device capable of facilitating the installation of the equipment.

This device is due to the importance of the correct calibration of the support points of the sales team. During the installation of the cold room, it is essential that its support points, made up of the load cells, support a similar weight between them.

If there is an imbalance, it could mean that the auto vending machine is crooked, overloading one or more of the weight cells, which could deteriorate the measurement or potentially damage said cell in the long run.

## 3. Results

Throughout the project, the development of an asynchronous execution Back End platform will be carried out, capable of performing data query and modification functions.

This platform will be able to gather the information generated by the different sales devices put into service, being able to store general information about them and their activities.

This platform also has the key function of updating the information stored in the control devices, such as the product sale price, Know Ice S.L. serial number and its location, the control device for its part will use this information to adequately communicate their status and perform their functions.

Another of the results obtained consists in the construction of functions capable of carrying out the complex tasks of controlling the device installed in the vending machine. These functions must update the information and communicate with the web platform, communicate with the payment platform and control the actuators distributed by the device.

Lastly, the cell leveler prototype is designed and completed, aimed at balancing the load cells of the sales team.

This device, powered from a USB port through the Raspberry Pi Pico, is connected to the load cells and, using a differential amplification scheme, measures the portion of the weight supported by each support point.

In this way, through an LCD screen installed in the device itself, the operator in charge of installing the refrigerating machine will be able to use the information offered by it for load balancing.

# Índice de la memoria

# INDICE DE FIGURAS

# INDICE DE CÓDIGOS

## EL ESTADO DEL ARTE:

En la actualidad, la forma más habitual de venta de hielo consiste en la instalación de grandes cámaras frigoríficas en el interior de edificios y centros de venta, en los cuales el cliente extrae los bienes deseados y los transporta a la línea de pago para su compra.

Las instalaciones que emplean este tipo de proceso de venta de hielo se ven obligadas a aumentar el coste de inversión para efectuar la climatización de sus centros, esto se debe a la existencia de las grandes cámaras frigoríficas que, al ser abiertas en repetidas ocasiones, se ven obligadas a actuar las bombas de calor para mantener la temperatura interior.

Esta acción aumenta el consumo eléctrico y la generación de calor en el edificio, el cual debe ser evacuado a su vez por los equipos HVAC de las instalaciones.

*Ilustración 1*

Una forma sencilla de anular este coste añadido es el de la descentralización del punto de venta de hielo, ubicando este en el exterior. Esto ya se realiza en diversos centros de venta, sin embargo, el método habitualmente empleado consiste en la habitación de la cámara frigorífica en el exterior del edificio y bloqueando el acceso a esta mediante cierres.

Esto obliga a los responsables de dicha máquina el realizar la apertura y cierre de los equipos para la extracción del hielo y su posterior venta en la línea de cajas.


*Ilustración 2*

Para facilitar este proceso, existe la posibilidad de producir máquinas de venta desatendidas, que realicen las funciones de almacenamiento y cobro de los productos.

Este producto, aunque no resulta muy común, ya ha entrado en el mercado por parte de varias distribuidoras.

El principal ejemplo es el de la empresa estadounidense *LEER*, la cual ofrece una serie de productos destinados a la venta de hielo, entre ellos se encuentra la gama *Ice Breaker*, orientada a la venta de hielo desatendida.


*Ilustración 3*

# Capítulo 1.- Introducción y planteamiento del proyecto

El objetivo del presente proyecto consiste en el diseño y fabricación de una máquina de autovending, es decir, de venta automática y no supervisada, destinada a la distribución de hielo al público.

La empresa española de distribución de hielo alimentario, conocida como Know Ice S.L., planea poner en funcionamiento un sistema de venta al público de hielo destinado al consumo. Siguiendo el esquema de almacenamiento en el distribuidor y recogida por el cliente, la empresa Know Ice S.L. quiere poner en funcionamiento un sistema distribuido de cámaras frigoríficas que sean capaces de hacer la función de venta de hielo, requiriendo del mínimo mantenimiento y monitorización.


*figura 1: ilustración de un diseño de ejemplo de las cámaras frigoríficas*

El diseño base, por lo tanto, consiste en la disposición de una cámara frigorífica sellada, diseñada para permanecer a la intemperie, que sea capaz de desbloquearse ante autorización externa y realizar un cobro proporcional de los bienes sustraídos.

A su vez, el dispositivo debe comunicarse con una estación de control, la cual lleva un registro de capacidad de los distintos dispositivos operativos. El fin es llevar a cabo un proceso de reabastecimiento de hielo óptimo, en función de la cantidad sobrante del contenedor y su ubicación física.

El diseño también requiere de la existencia de un sistema de control que autorice la apertura de la máquina.

Este sistema de autorización se realiza mediante una consulta bancaria, registrando los datos bancarios y solicitando confirmación de que la cuenta asociada al cliente posee un mínimo de liquidez mayor o igual al valor del contenido del congelador.

El diseño de la cámara frigorífica de venta automatizada exige que se aborden una serie de desafíos técnicos:

1- La cámara frigorífica deberá poseer una protección de calorifugado resistente a la intemperie, manteniendo un elevado grado de aislamiento térmico. Se ha de tener en cuenta que la cavidad frigorífica consta de distintas penetraciones para la inserción de dispositivos de medida y control, además de las aberturas frontales empleadas para la sustracción y abastecimiento del hielo. Este sellado será especialmente importante durante los días más calurosos, en los que la máquina podría estar expuesta al sol.

2- El sistema de control del dispositivo deberá discernir de forma precisa la diferencia en la cantidad de hielo antes y después de una transacción de bienes por parte de un usuario, de esta manera será capaz de realizar un cobro proporcional a la cantidad de hielo sustraída. Asimismo, deberá medir el contenido del congelador con el fin de advertir de la necesidad de reabastecimiento a la plataforma de monitorización y, de esta manera, que un operario pueda adelantarse a la necesidad y suplir la cámara frigorífica, antes de que se agote el producto.

3- El dispositivo de control en funcionamiento deberá contar con un sistema de conectividad y comunicación que permita la monitorización a distancia del estado de la máquina y su contenido. Es por lo tanto necesaria la integración de un sistema de telecomunicaciones y manejo de información para el correcto funcionamiento del sistema, más adelante se discutirán las distintas opciones existentes para abordar el presente problema.

4- El dispositivo deberá poder funcionar en paralelo a un sistema de autorización y control bancario que haga la función de verificación de fondos y más adelante realice la transacción bancaria necesaria. Por lo tanto, tanto el sistema de control del dispositivo como el sistema de cobro y autorización bancaria deberán poseer una línea de comunicación entre ambos.

5- Deberá seleccionarse un sistema de control programable capaz de realizar las funciones anteriormente descritas en los puntos 2, 3, 4 y 5. Este controlador al mismo tiempo tendrá que poseer la lógica de control para los distintos componentes de dispositivo de venta, siendo estos la apertura y bloqueo de puertas, monitorización de temperatura en el interior de la máquina y supervisión del estado de la máquina frigorífica. Por lo tanto, es imprescindible concretar el funcionamiento de los sistemas anteriormente descritos, ya que esto condicionará la selección del dispositivo microcontrolador.

6- Por último, será necesario el planteamiento de sistemas de seguridad y adquisición de permisos legales para la instalación de la máquina en el exterior (a resolver por Know Ice S.L.).

Será necesario además crear un Protocolo de Puesta en Marcha para la correcta instalación y puesta en funcionamiento de los dispositivos.

# Capítulo 2.- Descripción de las tecnologías (estado de la técnica)

Durante la primera etapa del desarrollo del proyecto, la falta de un esquema de prioridades limitó el avance de este debido al elevado grado de factores limitantes interconectados. Fue tras una larga etapa de investigación que poco a poco se definió tanto el alcance como los requisitos del proyecto, a medida que se iba allanando el terreno en búsqueda de técnicas que pudieran abordar los desafíos.

## 2.1 Aislamiento de la cámara frigorífica

Uno de los primeros puntos a discutir es el de la propia unidad frigorífica en sí.

Teniendo que ser capaz de operar en el exterior, la unidad frigorífica debe soportar las inclemencias del tiempo junto con elevados gradientes de temperatura en su aislamiento los días calurosos.

El diseño de la cámara frigorífica, por lo tanto, se debe realizar con la robustez en mente, con el fin de que esta perdure la máxima cantidad de tiempo y con el mínimo mantenimiento posible para que el proyecto sea rentable. De la misma forma, la selección y disposición de los mecanismos de control y medida deberán diseñarse con el fin de evitar el mayor número de penetraciones en el aislante posible, previniendo así que se deteriore la calidad del calorifugado.

Con estas condiciones en mente, se decidió modificar el cerramiento inicialmente definido como un pestillo con solenoide instalado en el interior de las compuertas, por un cierre electromagnético situado en el exterior de estas.

Por último, con el objetivo de equilibrar el peso de la cámara frigorífica sobre sus puntos de apoyo, se decidió ubicar el equipo frigorífico en la parte superior del congelador, equilibrando de esta manera el centro de masas del equipo.

El diseño de la cámara frigorífica se delegaría finalmente a la empresa de equipos frigoríficos portuguesa *Friconde.* El resultado de esta cooperación resultó en el diseño de dos modelos de cámara frigorífica de distintas capacidades.


*figura 2: Diseño preliminar cámaras frigoríficas (FRICONDE)*

El diseño de estos dispositivos cuenta con un exterior reforzado con placas plastificadas resistentes a impactos y a la exposición solar prolongada, un interior de poliuretano de 70mm, cierres exteriores y un compresor situado en la parte superior.

Ante el avance del proyecto, el diseño fue revisado con el fin de modificar los puntos de apoyo por unas sujeciones rígidas de altura regulable en lugar del diseño inicial de dos ruedas fijas y dos ruedas articuladas.

## 2.2 Control de contenido del congelador

Uno de los problemas fundamentales a resolver era el de la monitorización del contenido del refrigerador, tanto para el control de abastecimiento como para el cálculo de coste una vez realizada la compra.

En este campo de estudio se llegaron a abordar tres tipos de tecnologías distintas, cada una con distinto grado de precisión, complejidad y coste.

### 2.2.1 Tecnología óptica:

La primera técnica planteada para analizar el volumen de hielo sustraído fue la disposición de sensores ópticos e infrarrojos en el interior de la cámara frigorífica. Estos sensores se dispondrían en la parte superior del interior de la cabina para analizar la altura del contenido apilado de hielo.

Esta técnica, combinada con el análisis de imagen, podría comparar la disposición de los productos embolsados y calcular la cantidad de sacos de hielo que ha sido sustraída por el usuario.

Adicionalmente, el empleo de la tecnología óptica podría solventar el problema con la entrada de materiales extraños a la cámara. Dicha introducción de materiales podría suponer problemas a nivel de vandalismo (al introducir elementos ajenos al dispositivo con el objetivo de extraer hielo sin coste alguno), y a nivel legal; puesto que el dispositivo está expuesto al público y sin supervisión directa, debe estar protegido contra la posibilidad de que un sujeto se quede encerrado en su interior. En este sentido el sistema de análisis óptico de la cámara frigorífica podría evitar el cierre de las compuertas, facilitando la salida.


*figura 3: Cámara industrial Cognex In-Sight D900*

La imagen mostrada en la figura 3 corresponde a la cámara industrial modelo In-Sight D900 de la empresa Cognex. Sus avanzadas capacidades de aprendizaje Deep Learning y aplicaciones de Inteligencia Artificial, junto a su grado de protección IP67 la convierten en una pieza de tecnología candidata para su aplicación en el presente proyecto.

Sin embargo, esta tecnología fue descartada debido a una serie de ineficiencias que limitan su empleo.

La primera de estas limitaciones es la necesidad de emplear un sistema de control lo suficientemente avanzado como para poder realizar las tareas de computarización y procesamiento de imagen. Esto eleva innecesariamente el coste de producción, alejando el proyecto de su objetivo de flexibilidad y abaratamiento de costes.

Igualmente, la complejidad intrínseca de un programa capaz de analizar con precisión la imagen de la cámara frigorífica con el fin de estimar el volumen de hielo sustraído, podría prolongar la elaboración del proyecto por encima de la cota de desarrollo estimada, haciendo que la entrada de este en el mercado se posponga indefinidamente.

Además, la instalación de un sensor óptico en el interior de la cámara frigorífica añade la necesidad de emplear un dispositivo previsto con un grado de protección ante humedad, condensación y bajas temperaturas difícil de cumplir bajo los estándares industriales. La

necesidad añadida de realizar una penetración en la parte superior de la cavidad, que dañe el aislamiento de la cámara frigorífica, hace de esta tecnología una pobre opción.

Esta opción fue abandonada por otras con mayores niveles de fiabilidad y menor complejidad. No obstante, se propuso la adición de un sensor de altura acústico en la parte superior de la cavidad, para dar información adicional sobre el contenido del congelador que pueda contribuir al control de abastecimiento.

### 2.2.2 Tecnología RFID o UHF RFID:

La tecnología RFID o UHF RFID (Ultra High Frequency Radio Frequency Identification) consiste en la generación de un campo electromagnético de alta frecuencia dentro de un área local por parte de un elemento activo (lector), capaz de transmitir e inducir energía a unos dispositivos a través de una pequeña antena o bobina implementada en el interior de estos.

Los dispositivos RFID pasivos, en presencia de ese campo, son capaces de energizar el circuito integrado que poseen, el cual remite una señal de radiofrecuencia haciendo uso de la propia antena que actúa como inductor para energizar el circuito.

Por lo tanto, una vez energizado, el dispositivo RFID puede retransmitir un código predeterminado e identificativo dentro de un área pequeña, el cual es captado por una antena integrada en el propio lector. Una vez captada la señal y filtrada, un microcontrolador la decodifica y extrae el identificador único embebido en el circuito integrado del dispositivo RFID.

Esta tecnología, gracias a la simpleza del funcionamiento de sus elementos pasivos, y el hecho de que no requiera de un sistema autónomo de energía, como podría ser el de una batería, permite que se presente en formas más compactas como la de una tarjeta de crédito, o una etiqueta adhesiva.

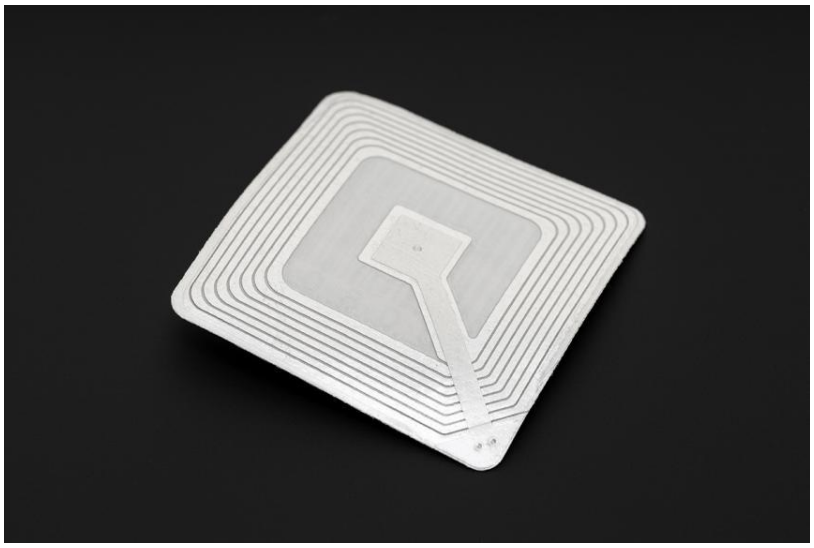


*figura 4: Ejemplo de un dispositivo RFID*

Esta tecnología, debido a su flexibilidad, se emplea habitualmente tanto en sistemas antirrobo en grandes superficies comerciales como en procesos industriales que requieran de un sistema de tarjetas identificativas, al ser estas más fiables que la lectura óptica de códigos de barras.

Debido a la forma de emisión de cada dispositivo RFID, el equipo lector es capaz de discernir entre las distintas señales por cada etiqueta identificadora dentro del área de acción. Esto permite el funcionamiento simultáneo de diversos identificadores RFID.

Teniendo esto en cuenta, la implementación de la tecnología RFID en nuestro proyecto, permite el recuento automático del número de bolsas de hielo alimentario en cualquier momento con precisión.

Para ello, se debe instalar un dispositivo lector RFID en el interior de la cámara frigorífica y el proveer a cada bolsa de hielo alimentario de una etiqueta adhesiva con tecnología RFID. De esta forma cada vez que el dispositivo de control requiera información sobre la cantidad de bolsas de hielo restantes, solicita al dispositivo lector que realice un proceso de lectura. Una vez realizado el sondeo, el lector RFID remite al microcontrolador los códigos recibidos dentro del área de acción, los cuales son identificados y contados por este.

Todo esto hace del RFID una tecnología fiable y versátil, fácil de adaptar a nuestras necesidades, sin embargo, fue descartada posteriormente debido a una serie de motivos.

El primero de ellos, consiste en la necesidad añadida de modificar la cadena de producción de los bienes de venta aguas arriba del punto de distribución. Es decir, en vez de integrar un nuevo sistema de venta al público para el mismo producto, una nueva etapa se integra dentro de la cadena de producción del hielo alimenticio, generando dos tipos de producto distintos, los lotes que poseen una etiqueta RFID y los que no.

Esta nueva etapa supone un coste añadido que no se previó en el lanzamiento del proyecto. De la implementación de la presente tecnología surge por lo tanto un coste añadido y un margen de error humano, debido al potencial abastecimiento de un lote sin tarjetas RFID. Dicho error supondría que el dispositivo de venta no sería capaz de detectar el número de productos sustraído durante la compra.

Como factor añadido, si se emplease esta tecnología, habría que blindar la cámara frigorífica ante interferencias electromagnéticas externas, lo cual podría suponer un riesgo a nivel de ciberseguridad y generar un coste añadido en la producción de los dispositivos.

Por esta serie de motivos, la tecnología RFID se rechazó en favor de una más sencilla de cara a la implementación.

### 2.2.3 Tecnología de medida directa de peso:

Por último, la tercera tecnología que se planteó para el desarrollo del dispositivo de venta consiste en la integración, dentro de la cámara frigorífica, de un sistema de medida directa del peso.

*figura 5: Células de carga industriales*

Inicialmente se estudió la posibilidad de instalar las células de carga en el interior del equipo. Es decir, la cavidad de la cámara frigorífica donde se introduce el hielo estaría desacoplada del resto del equipo. En su lugar se quedaría suspendida sobre los medidores de peso integrados en su interior. De esta forma, al ser las células de carga y medida la única unión entre la cavidad de la cámara frigorífica y el resto del equipo, éstas sólo medirían el peso del contenedor de hielo y su contenido.

Mediante este sistema, el margen de error se reduce considerablemente, al ser una medida cuasi directa de la masa del contenido de hielo de la máquina.

Sin embargo, debido a la dificultad añadida de realizar este diseño de cámara frigorífica, se rechazó por una opción más factible con el fin de abaratar el coste de producción de los equipos. El diseño final, por lo tanto, mantiene la cavidad de la cámara frigorífica acoplada al resto del equipo, como sería de esperar, e instala en su lugar las células de carga en la base del equipo, actuando éstas como punto de apoyo con el suelo.

Esta opción, aunque más factible en cuanto a la producción de los equipos por parte de *Friconde*, implica una dificultad añadida que repercute directamente en la precisión del sistema de medida y control.

Los equipos diseñados y propuestos por *Friconde* (representados en la *figura 2*), poseen las siguientes características:

| Model | Size X·Y·Z (mm) | Useful Volume | Weight | N° Ice Pack | Cooling (Gas) |
|---|---|---|---|---|---|
| C- 1315 | 1310 x 910 x 1600 | 1315 | 150 | 50/55 (10Kg) | R290 |
| C- 1870 | 1825 x 910 x 1600 | 1870 | 200 | 85/90 (10Kg) | R290 |

*figura 6: Características de los equipos frigoríficos propuestos por Friconde*

En el caso del modelo C-1315, se ha previsto una capacidad de carga de entre 500 y 550 Kg (50/55 bolsas de 10Kg) de hielo, mientras que el modelo C-1870 se estima que tiene una capacidad de carga máxima de entre 850 y 900 Kg (85/90 bolsas de 10Kg).

Al mismo tiempo, en el caso del primer modelo, el peso del equipo en vacío se estima que será en torno a los 150Kg, mientras que el modelo C-1870 asciende hasta 200Kg.

Por lo tanto, el caso más restrictivo, y en el que vamos a centrar el estudio del presente proyecto será el del desarrollo de un sistema de control para el modelo C-1870.

## 2.3 Protocolo de monitorización a distancia del contenido de los dispositivos:

Uno de los requisitos para la realización del proyecto es el de la implementación de un servicio de telecomunicaciones que recoja y centralice las notificaciones de los distintos dispositivos de distribución para controlar el contenido y reponer el material de venta cuando fuera necesario. La primera línea que se planteó fue a través de la red de telecomunicaciones Sigfox.

SigFox, que en España opera bajo el nombre de Unabiz, es una empresa de telecomunicaciones especializada en servicios de IoT (Internet of Things). Esta empresa ofrece servicios LPWAN (Low Power Wide Area Network) para soluciones enfocadas en el IoT, basándose en el uso de tecnología 0G.

La red de SigFox consiste en una distribución de antenas que ofrecen un servicio de banda estrecha en un área amplia, cubriendo gran parte del territorio europeo con la menor infraestructura.

Esta red puede ser aprovechada por dispositivos que requieran de conectividad desde puntos lejanos, sin necesidad de emplear redes locales de comunicación convencional como son el 3G y 4G, y eliminando la necesidad de emplear comunicación satelital cuando no se dispone de estas redes.

A su vez la red 0G de SigFox se encuentra estructurada en un servicio en nube al que se puede acceder para la consulta y emisión de datos a través de otras aplicaciones integradas en API (Interfaz de Programación de Aplicaciones) o consultada desde la plataforma puesta a disposición de los usuarios.

Sin embargo, con tal de cubrir un área amplia, la tecnología 0G se limita a ofrecer un servicio de telecomunicaciones de banda estrecha, pudiendo transportar paquetes de datos de muy poco volumen.

El servicio ofrecido por SigFox permite la emisión, por parte de los dispositivos de IoT distribuidos en su red, de paquetes de datos (Payloads) de hasta 12 Bytes. Además, estos se envían un máximo de 140 veces al día (aproximadamente cada 10 minutos). Esto es para evitar la saturación de la red. Al mismo tiempo, permite la recepción de paquetes de

datos de 8 Bytes con un máximo de 4 Bytes cada 24 horas, este paquete es enviado por el servicio en nube de SigFox, a petición de la API o el usuario en control del dispositivo de IoT y sirve para proveer a éste de un feedback o información adicional si esto fuera necesario.

Es por ello por lo que se trata de un servicio destinado a dispositivos de IoT que no requieren emitir un volumen elevado de datos. Es empleado en entornos de industria, instalaciones de energía, centros de logística, etc. En los que se necesite manejar variables sencillas que no requieran mucha velocidad de control.


*figura 7: estructura de servicio SigFox*

Debido a todo esto, de cara al proyecto, la tecnología 0G de SigFox, ofrece un servicio adecuado para la monitorización de los dispositivos de venta, a pesar de sus limitaciones.

Al abarcar un área tan amplia del terreno, los dispositivos de venta pueden situarse en ubicaciones más lejanas para ofrecer el servicio de venta de hielo alimentario. Teniendo que enviar información reducidas veces al día, puede codificarse en 12 Bytes el volumen de hielo remanente, la temperatura interior y exterior del equipo, el número de ventas… de la misma forma, a través la API de Know Ice S.L., la plataforma de control es capaz de enviar la orden de puesta en servicio del equipo, ordenar su desconexión, o actualizar el precio por volumen del hielo mediante la emisión de paquetes de 8 Bytes de datos.

Adicionalmente, la implementación del sistema de telecomunicación de SigFox es un proceso accesible, tratándose en algunos casos de una comunicación entre el microcontrolador que se quiera conectar, con un dispositivo de radio ofrecido por la propia empresa o empresas afiliadas a esta. Este dispositivo de radio emplea una sencilla comunicación I2C con el microcontrolador, a través de los puertos RX-TX que éste disponga.

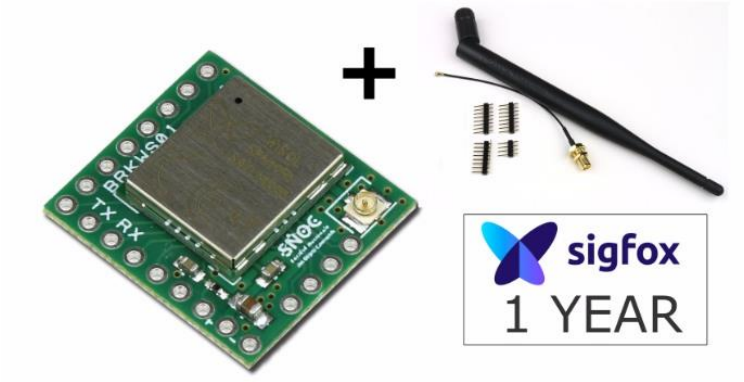



*figura 8: Imagen del kit Sigfox Breakout board BRKWS01*

En la *figura 8* se representa el que hubiera sido el Sistema de comunicación SigFox empleado para el desarrollo del prototipo. Al emplear un protocolo de comunicación tipo I2C no hubiera supuesto un factor limitante a la hora seleccionar un microcontrolador para el control del dispositivo de venta.

Sin embargo, pese a sus numerosas ventajas, el sistema de telecomunicación 0G de SigFox se descartó más adelante en el desarrollo del proyecto, y fue sustituido por un protocolo de comunicación convencional basado en redes locales 3G y 4G. El motivo de abandonar los servicios de SigFox es debido a la redundancia que supone el uso de la red 0G en paralelo a la comunicación convencional, la cual es imperante para el funcionamiento del servicio del terminal de pago que emplea el dispositivo para la realización de ventas.

## 2.4 Plataforma de pagos

Como es de esperar, el dispositivo de venta de hielo requiere de un terminal de pago para realizar la función de autorización bancaria y cobro post venta.

Para incorporar las funciones de la plataforma de pago, se contactó con distintas empresas que ofrecen servicios y dispositivos destinados a sistemas de venta no supervisados, como el del proyecto.

El dispositivo debe funcionar en paralelo a un sistema de autorización y control bancario que haga la función de verificación de fondos y más adelante transacción bancaria cuando esta sea necesaria. Por lo tanto, tanto el sistema de control del dispositivo y el sistema de cobro y autorización bancaria deben poseer una línea de comunicación entre ambos.

En primer lugar, se contactó a la empresa SEPALO, quienes ofrecen un servicio de plataforma de pago y terminal bancario para quioscos y cajeros, siendo posible adaptarlo de cara al proyecto, aunque con ligera dificultad. Sin embargo, debido a complicaciones

en la relación entre SEPALO y la empresa promotora de este proyecto, la cooperativa terminó siendo disuelta por parte de la empresa de servicios de gestión de ventas.

El siguiente intento de cooperación fue con myPOS, empresa orientada a la creación de servicios integrales de pago. Tras varias semanas de comunicación con los encargados del servicio, se postuló la posibilidad de integrar su solución para terminales de puntos de venta, orientado a máquinas expendedoras, aparcamientos y ventas de entradas.


*figura 9: myPOS Integra para TPV desatendido*

Los dispositivos de myPOS funcionan a través de sistemas integrados en máquinas virtuales. Para hacer uso de estos programas se necesita un servicio API que se comunique con la plataforma de myPOS, siendo este un programa que debe ejecutarse desde un sistema operativo como Linux o Windows.

Este último factor obligó a replantear la cooperación con la empresa de myPOS, ya que la necesidad de ejecutar el programa de control desde una plataforma con un sistema operativo embebido se alejaba tanto de los conocimientos del ejecutor del proyecto, como del alcance propuesto inicialmente para la realización de este, elevando el coste y dificultando innecesariamente la implementación del sistema.

Debido a dichos motivos, se decidió renunciar a la cooperativa con myPOS en favor de la empresa PAYTEF.

PAYTEF se trata de una empresa que ofrece un servicio de pasarela de pago flexible, quien a su vez coopera con la empresa PAX, desarrolladora de terminales de pago y datáfonos. Entre los productos de PAX, PAYTEF ofrece una gama de terminales orientado a los servicios de venta desatendidos.

Tras contactar con el servicio de PAYTEF y habiendo puesto en común las características y alcance del proyecto, PAYTEF dispuso un servicio de simulación de venta y un terminal de pago destinado al desarrollo de prototipos, modelo A50.


*figura 10: Terminal de pago modelo A50 de PAX*

El terminal de pago A50 de PAX, consiste en un dispositivo android que, a través de la plataforma back End desarrollado por Paytef, puede comunicarse con aquellos dispositivos que posean previa autorización y cumplan con los protocolos de seguridad de la plataforma.

A través de la plataforma Back End, un dispositivo de control conectado a la red es capaz de enviar una petición de autorización al terminal de pago. Haciendo uso de protocolos de comunicación HTTP, el dispositivo de control puede enviar peticiones tipo POST a la dirección URL asociada al terminal de pago en uso.

Estas peticiones POST van adjuntas a un paquete de datos en formato JSON con estructura de objeto. Un objeto es una entidad propia del entorno JavaScript; esta es similar en estructura a un array, sin embargo, en vez de enumerarse las posiciones, estas van asociadas a propiedades.

Por lo tanto, mediante un objeto se puede enviar un paquete de datos completo con la información que requiera el terminal de pago o el microcontrolador que esté manteniendo una comunicación con éste. De esta forma se facilita la transmisión de información con respecto a otros formatos. Es por eso por lo que los objetos son la forma más habitual de comunicación dentro del entorno de servidores y clientes.

La posibilidad de externalizar el protocolo de venta al conjunto de servicios ejecutados en la nube y el terminal de Android permite reducir los requisitos de capacidad y procesamiento del dispositivo controlador. Esto anula la necesidad de un dispositivo capaz de ejecutar en una máquina virtual el sistema operativo de un ordenador, mientras emplea los puertos accesibles a este para el control de los actuadores, en su lugar se utiliza un dispositivo de control monotarea más sencillo, eficaz y de menor coste, con el único requisito de que pueda realizar comunicaciones siguiendo un protocolo web.

El dispositivo A50 sin embargo resultó no ser el indicado para la instalación en el prototipo final de autovending. En su lugar, se escogió el terminal de pago puesto a disposición por parte de Paytef y la empresa PAX, modelo IM30.


*figura 11: Terminal de pago modelo IM30 de PAX*

No obstante, con el propósito de llevar a término el prototipo del presente proyecto, y ya que ambos modelos de terminal de pago funcionan siguiendo los mismos principios básicos, se hizo uso del dispositivo A50, a través del entorno de simulación que ha sido proporcionado para generar la estructura básica del programa.

## 2.5 Elección de controlador

Durante la primera etapa del proyecto, la elección de un dispositivo microcontrolador que realizara las funciones necesarias para la correcta operabilidad del prototipo fue un proceso que sufrió numerosas modificaciones hasta concretarse una opción viable.

En un principio se planteó el empleo de un microcontrolador básico como la gama MSP430Fxxx de Texas Instruments o MC 68HC908xxx de Motorola. Sin embargo, ante la incompatibilidad que presentaban frente a algunos de los módulos del proyecto, se decidió abandonar la propuesta por un centro de control más capaz.

En esta fase del proyecto, se vio la necesidad de ejecutar un sistema operativo como Linux, Windows 7 o superior. Con un sistema semejante, sería posible llevar a cabo las funciones necesarias para la operabilidad de la pasarela de pago.

Por este motivo, se planteó la utilización de ordenadores industriales. Éstos consisten en ordenadores con una mayor capacidad y robustez, pudiendo funcionar de manera independiente para el control centralizado de procesos industriales, siendo mucho más fiables que los ordenadores convencionales.

Sin embargo, esta elección resulta en una aplicación innecesaria de una tecnología poco convencional, excediendo los requisitos de nuestro dispositivo. Pese a la alta variedad de opciones presentes en el mercado, la aplicación de un ordenador industrial en su forma más simplificada, como podría ser el modelo de Phoenix Contact 1130669, supone un elevado coste para el uso de un hardware con unas propiedades significativamente superiores a las necesitadas por este proyecto.

*figura 12: Ordenador industrial modelo 1130669 de Phoenix Contact*

En lugar del uso de ordenadores industriales como el descrito anteriormente, se propuso sustituir este por una placa programable de la familia Raspberry. Siendo significativamente más accesible comercialmente, y facilitando el manejo de entradas y salidas lógicas, la Raspberry Pi es capaz de ejecutar un sistema operativo como Windows en su versión ARM.

Esto genera una diferencia en el tipo de procesamiento con el que se ejecutan los programas; el ordenador industrial descrito con anterioridad emplea un procesador tipo x86. Es decir, el equivalente a un ordenador convencional. Mientras tanto, las placas programables de Raspberry hacen uso de un procesador similar al de un smartphone, conocido como ARM (Advanced RISC Machine, refiriéndose RISC a Reduced Instruction Set Computer).

Esto supone una diferencia en la arquitectura interna del propio sistema operativo que se ejecute en él, limitando a veces el tipo de programas que pueden ser procesados en este microcontrolador. No obstante, existe la posibilidad de que una placa ARM como la Raspberry Pi ejecute un sistema operativo como Windows 11, debido a la existencia de una versión que ha sido migrada específicamente a este sistema.

Gracias a esto, en teoría, la placa controladora de Raspberry Pi podría realizar las funciones de control del prototipo del sistema de venta de hielo, al tiempo que ejecuta la plataforma de pago de MyPOS para manipulación del terminal de pago. Por desgracia, la empresa de myPOS no fue capaz de confirmar la viabilidad de la propuesta, ya que desconocen si el programa es capaz de ejecutarse en un dispositivo ARM.

Poco después, se cesó la relación con la empresa de myPOS en pos de la plataforma de pago ofrecida por Paytef. Debido a las características del funcionamiento de su plataforma y terminales de pago, el sistema de control podía volver a simplificarse, esta vez en la forma de un microcontrolador más convencional, como es en este caso la Raspberry Pi Pico W.

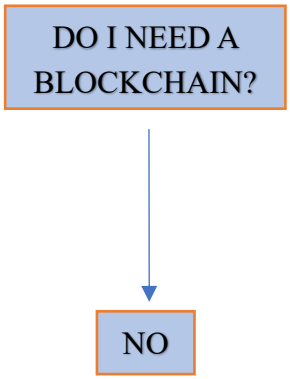
Las características del terminal de pago ofrecido por la empresa de Paytef y su afiliada, PAX, permiten que se realicen las operaciones del terminal de pago a través de la plataforma API de Paytef, mediante la emisión de peticiones HTTP. Por este motivo, el único requisito para su manipulación desde el sistema de control es la posibilidad de realizar este tipo de protocolo web.

Por este motivo se ha decidido hacer uso de la placa microcontroladora Raspberry Pi Pico W, gracias a su alta capacidad para el control de sistemas unifilares (es decir, ejecuta programas síncronos) y la capacidad de realizar peticiones web gracias a su módulo wifi integrado. Siendo además altamente accesible a nivel comercial, la Raspberry Pi Pico W es el candidato perfecto para la realización del prototipo de este proyecto.


*figura 13: microcontrolador Raspberry Pi Pico W*

# Capítulo 3.- Descripción del modelo desarrollado

Durante la primera etapa en la realización del proyecto, debe plantearse la posibilidad de aplicar tecnologías vanguardistas, focalizadas en la transmisión de información con alta seguridad a través de la red en lugar de la convencional comunicación HTTP. Para tomar dicha decisión, se emplea el siguiente esquema de decisiones aplicado a proyectos:




Una vez tomada la decisión, se aplica la tecnología de comunicación web al proyecto a través de la plataforma Node.js.

A continuación, se expone el funcionamiento del modelo final.

## 3.1 API:

La API de simulación se ejecuta desde la lanzadera de Visual Studio Code, dentro de la carpeta KNOW_ICE se encuentran una serie de archivos de JavaScript necesarios para el funcionamiento de la API.

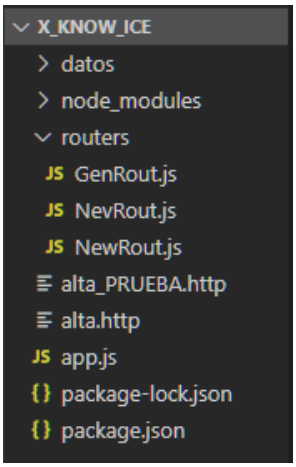



*figura 14: documentos JS*

El fichero principal que se muestra en la figura 14 es el definido como "*app.js*". Este centraliza las funciones del servidor y define los enrutamientos que serán empleados por los documentos de la carpeta routers.

En la carpeta routers se expanden las funciones del servidor, definiendo las respuestas y procesos a realizar una vez se reciba una petición por parte de un cliente a una dirección determinada.

También se encuentran archivos .http, estos serán empleados para simular las acciones de un hipotético cliente del servidor, emitiendo peticiones POST, PUT, DELETE y GET.

Por último, se encuentran dos archivos .json, estos definen las dependencias para el funcionamiento de nuestro programa y se generan automáticamente.

### 3.1.1: app.js

A continuación, se explica brevemente el documento *app.js* con el fin de ilustrar el funcionamiento de las peticiones web recibidas por el servidor

```
const { Console } = require('console');
const express = require('express');
const app = express(); //funcion que nos permite crear una aplicacion de express

const NevRout = require('./routers/NevRout.js');
app.use('/api/knowice/dispositivos', NevRout.routerNEV);
const GenRout = require('./routers/GenRout.js');
app.use('/api/knowice/datagen', GenRout.routerGEN);
const NewDisp = require('./routers/NewRout.js');
app.use('/api/knowice/ID/new', NewDisp.routerNEW);



//implementamos la primera ruta con express
app.get('/', (req, res) => {
  res.send('Servidor local Know ICE');
});


const PUERTO = process.env.PORT||3000; //puerto 3000 en la red local

app.listen(PUERTO, () => {
  console.log(`el servidor está escuchando en http://localhost:${PUERTO}`);
  console.log(`http://localhost:${PUERTO}/api/knowice/ID/new`);
  console.log(`http://localhost:${PUERTO}/api/knowice/datagen`);

});
```

*Código 1: app.js*

La primera acción realizada en el archivo *app.js*, es la inclusión de las librerías principales que se van a usar; la librería "console" provee al programa, de forma explícita, el equivalente a la consola de depuración (debugging) que, a su vez, está implícita en un servidor web. La siguiente librería que se incluye es la "Express" (incorporada dentro del objeto llamado app), esta librería facilita el manejo de peticiones tipo http, y es empleada para responder y procesar las peticiones.

Posteriormente, se importan los archivos *.js* ubicados dentro de la carpeta de routers mediante la función *require*, una vez importados, se establece una ruta sobre la que responderán los distintos routers (p.ej: http:……/api/knowice/dispositivos). Por lo tanto, cada petición enviada a una dirección derivada de las definidas, serán remitidas a su router correspondiente y procesado por el mismo.

Las rutas también pueden definirse en el propio documento de app.js, como se puede ver en la línea 13 de código. Se define una sencilla respuesta a la recepción de una petición de tipo *get* a la ruta raíz del servidor. Respondida con el texto "Servidor local Know ICE".

Por último, se define el 3000 como puerto de funcionamiento del servidor dentro de la red local, y se inicia la escucha del servidor mediante la función *app.listen* , atendiendo a las peticiones recibidas por puerto 3000.

Debido a que se ejecuta el servidor de forma local en una red doméstica, no funcionará a menos que se esté conectado a ella. Por lo tanto, para acceder al servidor se hace a través de la ruta http://localhost:3000/ en el ordenador (el cual cumple la función de servidor) o

la ruta http://192.168.1.11:3000/ desde cualquier dispositivo ajeno conectado a la red, siendo 192.168.1.11 la dirección IP local del dispositivo que ejecuta el programa.

Para comprobar el estado del servidor, se realiza una petición de tipo *get* mediante la búsqueda de la dirección raíz, haciendo uso de la función descrita en la línea 13 del documento *app.js*.

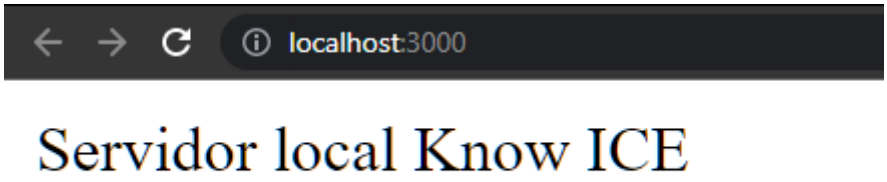



*figura 15: Respuesta recibida en la dirección http://localhost:3000/*

### 3.1.2: GenRout.js

El router GenRout.js centraliza las funciones de consulta y modificación de datos generales de los dispositivos de venta, correspondiendo a la ruta http://localhost:3000/api/knowice/datagen. Para la creación de un nuevo router, se debe importar en el nuevo documento la librería "Express", y mediante esta, crear una constante que posea las propiedades del router a través de las siguientes líneas de código.

```
const express = require('express');
const routerGEN = express.Router();
```

*Código 2: Creación y asignación del router en GENrout.js*

Dentro del propio documento se accede a las propiedades y funciones del router mediante el objeto asignado con nombre *routerGEN*.

Por último, para poder hacer uso del archivo GenRout.js, se añade al final del documento la siguiente línea de código:

```
module.exports.routerGEN = routerGEN;
```

*Código 3: exportación del módulo routerGEN, GenRout.js*

De esta manera, podrá ser importado desde la aplicación principal, app.js para su uso como enrutador.

#### *3.1.2.1: consultas al servidor*

Haciendo uso de la ruta http://localhost:3000/api/knowice/datagen/all, se puede obtener la información general de todos los dispositivos dados de alta por la plataforma, contenidos dentro del objeto *dataGen* en el servidor, gracias al siguiente código:

```
routerGEN.get('/all', (req,res) => { //info general de todos los dispositivos
    res.status(200).send(dataGen);
});
```

*Código 4: Petición GET de información de todos los dispositivos, GenRout.js*

Esta búsqueda devuelve la siguiente información:


*figura 16: resultado consulta http://localhost:3000/api/knowice/datagen/all*

Se ha de tener en cuenta que las peticiones *get* emitidas al servidor local, devuelven un paquete de información en formato JSON. El buscador web, por lo tanto, representa la información en formato de texto. La presencia de un entorno de programación Front End, podría exponer dicha información de una manera más ilustrativa, permitiendo además modificar o filtrar la información de los dispositivos. Sin embargo, ya que la programación del entorno Front End queda fuera del alcance del presente proyecto, se emplea el programa en su estado Back End para realizar estas funciones y consultas desde las direcciones URL correspondientes.

Las direcciones URL se subdividen en distintos segmentos; por ejemplo, la dirección https://www.knowice.es/inicio/ aporta la información sobre el protocolo web (https), el subdominio (www), el dominio (knowice), el dominio de nivel superior, la cual se traduce a la dirección IP final del DNS (.es) y, por último, el *path* (/inicio/), el cual indica la ruta dentro de la dirección IP descrita por el dominio.

Adicionalmente se pueden aportar parámetros de búsqueda, también conocidos como parámetros *query*, al final de la dirección URL, para ordenar y filtrar la información devuelta por el servidor. Por ejemplo, si se quisiera solicitar la información de los dispositivos pertenecientes a la empresa Know Ice S.L., ordenado en función del número de ventas, se solicitaría, añadiendo al final de la dirección URL las siguientes propiedades: "?empresa=knowice&ordenar=ventas"

De esta manera, si se quisiera filtrar la información devuelta por el servidor en función de la empresa propietaria del dispositivo, se añade dicha empresa al final de nuestra dirección URL. Esta petición es procesada por la siguiente línea de código:

```
routerGEN.get('/:empresa', (req,res) => { //ruta para cada empresa
  const empresa = req.params.empresa;
  const resultados = dataGen.filter(disposi => disposi.empresa === empresa);

  if(resultados.length === 0) {
    return res.status(404).send(`No se encontraron dispositivos de ${empresa}.`);
  }

  res.send(JSON.stringify(resultados.sort((a,b) => a.KI_ID - b.KI_ID)));
});
```

*Código 5: petición GET con filtro por Empresa, GenRout.js*

La consulta a la dirección (…)api/knowice/datagen/Empresa1 devuelve el siguiente resultado:


*figura 17: resultado consulta http://localhost:3000/api/knowice/datagen/Empresa1*

Por último, añadiendo la siguiente función se podría realizar un segundo filtrado del contenido del objeto *dataGEN*, consultando dispositivos en función de su ID de Know Ice:

```
routerGEN.get('/:empresa/:ki_ID', (req,res) => { //ruta para cada empresa
  const empresa = req.params.empresa;
  const ki_id = req.params.ki_ID;
  const resultados = dataGen.filter(disposi => disposi.empresa === empresa && disposi.KI_ID ==
ki_id);

  if(resultados.length === 0) {
    return res.status(404).send(`No se ha encontrado el dispositivo ${ki_id} de ${empresa}.`);
  }

  res.send(JSON.stringify(resultados));
});
```

*Código 6: petición GET con filtro por Empresa e ID de Know Ice, GenRout.js*

Hay que destacar que el filtro de consulta sobre el ki_ID, se compara empleando un doble signo de igualdad, en vez de hacerse mediante un triple signo igual. Esto es debido a que la comparación empleando "===" exige que ambos valores que se comparan sean del mismo formato, en el caso de la empresa se compara texto con texto. Sin embargo, al comparar el ID, la consulta web interpreta los parámetros URL como texto, mientras que los valores del atributo KI_ID dentro del objeto dataGen se pueden encontrar en formato numérico. Por este motivo la comparación se realiza mediante el protocolo "==", el cual no requiere la igualdad de formato y valor conjuntos.

La consulta a la dirección (…)api/knowice/datagen/know ICE/2 devuelve el siguiente resultado:

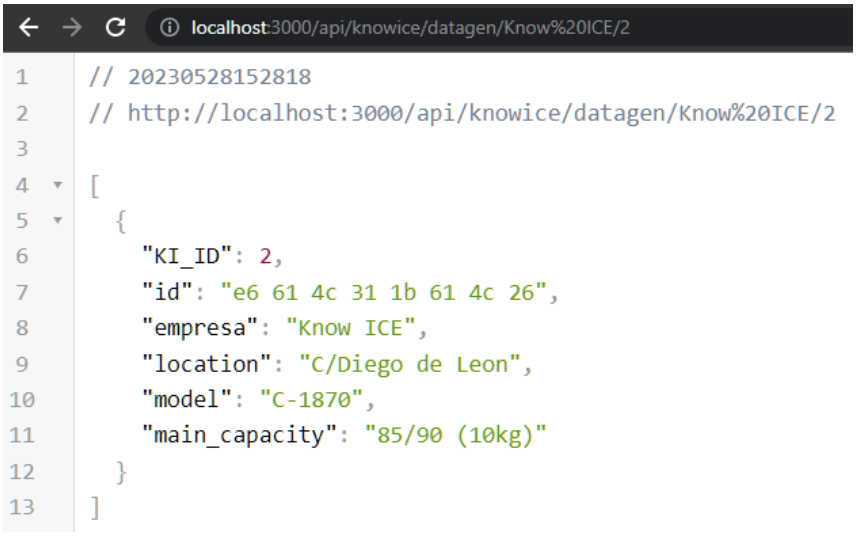



*figura 18: resultado consulta http://localhost:3000/api/knowice/datagen/Know%20ICE/2*

Por otro lado, el código de consulta que se ha representado posee un control de filtrado en el caso de que la búsqueda no obtenga resultados. En dicho caso el servidor nos devuelve el siguiente mensaje.

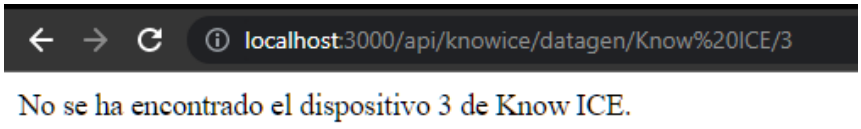



*figura 19: resultado consulta http://localhost:3000/api/knowice/datagen/Know%20ICE/3*

Con este código se puede gestionar la mayoría de las consultas que pueden realizarse desde la plataforma de Know Ice S.L. Adicionalmente, se pueden añadir las funciones necesarias para implementar un sistema de control de acceso. Esto permite consultar sólo los dispositivos dados de alta por una empresa en caso de pertenecer a terceros. Mientras tanto, Know Ice S.L. tiene acceso a la información general de todos los dispositivos dados de alta para fines de mantenimiento y reabastecimiento de los dispositivos.

Sin embargo, sólo se han contemplado hasta ahora las consultas al servidor, en forma de peticiones de tipo *get*. Por lo tanto, quedan por definir las funciones encargadas de gestionar el alta de los dispositivos, su baja, y las modificaciones que se les quisiera hacer a su información general.

#### 3.1.2.2: base de datos

En el caso particular de este proyecto, la base de datos se ha gestionado como un objeto vacío, contenido como una variable dentro del servidor. En la aplicación final del proyecto, junto con la implementación en el servidor de Know Ice S.L., se hace uso de la base de datos ofrecida por el proveedor de los servicios web con el fin de ganar robustez en el servicio.

La solución empleada en este proyecto no aporta una seguridad frente a apagones en el servidor, lo cual haría que los datos se perdieran. Con el objetivo de mantener la información de los dispositivos, la API tendría que cooperar con una plataforma de manejo y almacenamiento de datos basada en SQL. Sin embargo, con el objetivo de facilitar la realización del proyecto se ha optado por almacenar la información en la memoria del servidor.

Para almacenar la información, primero se genera una variable tipo *array* vacía de contenido en la parte superior del código, dentro de GenRout.js.

```
const dataGen = [

  ]
```

*figura 20: asignación de la variable dataGen, GenRout.js*

#### 3.1.2.3: middleware

Las peticiones HTML recibidas o enviadas por el servidor, almacenan su contenido en un paquete de datos traducido a formato JSON (JavaScript Object Notation). Este es un formato de texto ligero empleado en el entorno de comunicación web, que permite el envío de objetos.

Para su procesamiento, se debe de incluir un *middleware* en el servidor. Esto es una función software que actúa como un agente intermediario entre un cliente y servidor. De esta manera es capaz de modificar la información intercambiada entre ambos dispositivos, igualando su formato a pesar de que posean naturalezas distintas. Así se facilita el intercambio de información sin importar las características del lenguaje empleado en el origen.

Para implementar en el código el *middleware* es obligatorio iniciarlo de manera previa a las funciones dedicadas a peticiones POST, PUT y DELETE. De no ser así las funciones no serán capaces de procesar el cuerpo de la solicitud recibida.

```
routerGEN.use(express.json());
```

*Código 7: : inicialización middleware, GenRout.js*

### 3.1.2.4: alta de dispositivos

Dar de alta a los equipos es un proceso que consiste en dotarlos, en una base de datos, de su información general. En el caso de este proyecto, la información genérica de cada dispositivo consiste en su número identificador de Know Ice S.L. ("KI_ID"), el ID único implícito de cada dispositivo ("id"), la empresa propietaria ("empresa"), la ubicación en la que se ha instalado el equipo ("location"), el modelo de cámara frigorífica ("model"), y su capacidad teórica ("main_capacity").

```
{
    "KI_ID": "",
    "id": "e6 61 4c 31 1b 61 4c 25",
    "empresa": "Know ICE",
    "location": "C/Moratin",
    "model": "C-1870",
    "main_capacity": "85/90 (10kg)"
}
```

*figura 21: ejemplo de información general de un dispositivo, GenRout.js*

En la figura 21, se puede observar el modelo empleado para ejemplificar la información relativa a los dispositivos. En el modelo final, podrían incluirse nuevos valores como el precio del producto por unidad de peso, el valor umbral de notificación por falta de hielo y aspectos más técnicos como la imagen codificada del logo de la empresa propietaria o publicidad a representar en la terminal de pago.

El proceso de alta de los dispositivos se coordina con la información aportada por el propio equipo. Una vez energizado y conectado a la red, el dispositivo de control de la máquina frigorífica realiza una petición tipo *get* al servidor, solicitando la información general de los equipos dados de alta. Una vez obtenida, si este no localiza la información de un equipo asociado a su ID, publica en una dirección URL de nuestro servidor su propio ID.

En esta dirección se puede consultar el ID del equipo recién conectado y, mediante la interfaz Front End de Know Ice S.L., darlo de alta. Este proceso es explicado con mayor profundidad más adelante, a través del código de control del propio dispositivo de la RaspBerry Pi Pico W.

En el caso específico de este proyecto, al no contar con una interfaz Front End dedicada a facilitar el alta de los equipos, se debe emitir la petición a mano. Esto se realiza mediante la extensión de Visual Studio Code, REST Client. Gracias a esta aplicación, se puede generar un archivo http con la capacidad de simular las peticiones de un cliente ajeno a nuestro ordenador.

A continuación, se muestra el código (figura 22) del archivo alta.http empleado para solicitar el alta de un equipo teórico recién instalado.

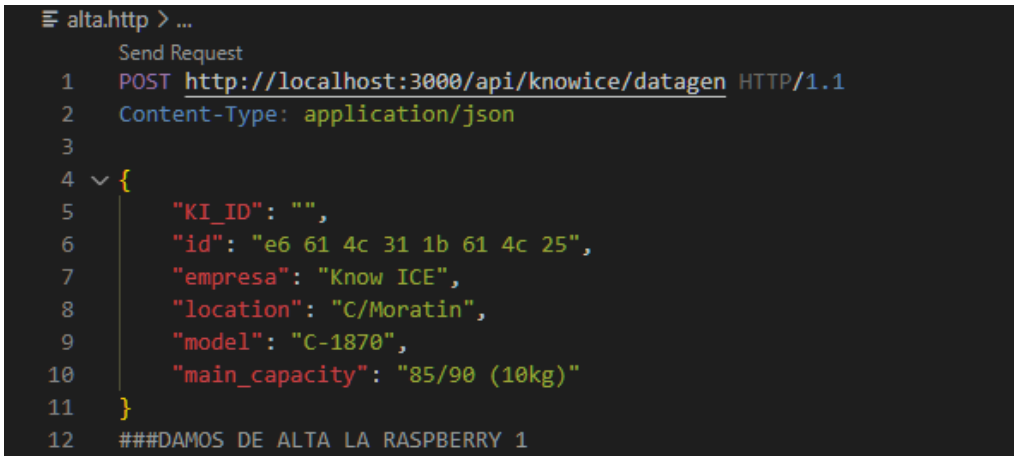



*figura 22: Payload de solicitud de alta de equipos, alta.http*

Gracias a la extensión REST Client, generar un código dedicado a la emisión de peticiones POST y PUT se simplifica significativamente, anulando la necesidad de emplear un segundo dispositivo para ello.

Para crear la petición, basta con crear un documento con terminado en ".http" dentro de la carpeta de Visual Studio Code. Una vez ahí, se escribe en la primera línea el tipo de protocolo que se quiera emplear.

El recurso POST es empleado para la generación de nuevos bloques de datos. La API encargada de procesar dichas peticiones, una vez reciba un recurso identificado como tal, generará un nuevo paquete en su base de datos para almacenar el contenido del cuerpo del mensaje una vez procesado.

Por otro lado, el recurso PUT, se emplea para modificar información ya existente. La propia API, una vez recibe la petición, se encarga de localizar el paquete de datos referenciado por la petición HTML con el objetivo de realizar las modificaciones necesarias, o incluir nuevos atributos

A continuación, se muestra el código encargado de procesar el alta de los equipos:

```
//POST para dar de alta un nuevo disp
routerGEN.post('/', (req,res) => {
  let New_data = req.body;
  //añadimos una validación un poco rudimentaria para asegurarnos de que tiene todo lo necesario
  if(New_data.KI_ID != null && New_data.id != null && New_data.empresa != null
    && New_data.location != null && New_data.model != null && New_data.main_capacity != null)
    {
      let indice = dataGen.findIndex(nevera => nevera.id == New_data.id);
      if(indice<0){
        New_data.KI_ID = 1;
        for(i=0; i<dataGen.length; i++){
          if(New_data.KI_ID <= dataGen[i].KI_ID){
            New_data.KI_ID = dataGen[i].KI_ID + 1;
          }  //siempre que se de de alta uno nuevo su ID será 1 número más que el más alto,
        }    //si se borra uno ese hueco no se llenará, de esta forma no se pierde trazabilidad
de las IDs antiguas
      }
      else{
        res.write(JSON.stringify(`Esta ID ya esta dada de alta -> `));
        res.write(JSON.stringify(dataGen[indice]));
        return res.status(404).end();
      }
      dataGen.push(New_data); //es un array recuerdas
      res.status(200).send(JSON.stringify(dataGen)); //validamos
    } else
```

```
  {
    res.write(JSON.stringify(`Información incorrecta o incompleta`));
    return res.status(400).end();
  }

});
```

*Código 8: alta de equipos, GenRout.js*

En el caso del ejemplo mostrado en el código 8, la API procesa una petición de tipo *post* recibida a la ruta raíz de GenRout.js (http://localhost:3000/api/knowice/datagen).

El protocolo HTTP distingue dos cuerpos principales en el procesamiento de las peticiones; el primero de ellos es el cuerpo *req* (request). Este contiene la información sobre la petición enviada por el cliente, y la naturaleza de ésta. La otra parte consiste en el cuerpo *res* (response), este es el mensaje que se devuelve al cliente desde el servidor, y sirve para redirigir el contenido de la respuesta a la petición y el código HTTP.

La emisión del código de respuesta es imprescindible para el correcto funcionamiento de la comunicación web. Dichos códigos representan un feedback al cliente, anunciando si la petición ha sido recibida con éxito o ha sucedido un error. Los códigos HTTP que refieren a peticiones realizadas con éxito, oscilan entre el 200 y el 299, siendo el más habitual el código 200. Por otro lado, de haberse producido un error en la emisión o procesamiento de la petición, el servidor respondería con un código comprendido dentro del margen de 400 y 499, siendo el 404 el más conocido y empleado.

Hasta la recepción del código de respuesta HTTP por parte del cliente, la línea de comunicación se queda abierta, pudiendo derivar en errores o bloqueos del programa.

En este caso, al detectar la API la entrada de una petición tipo *post* recibida a la ruta raíz de GenRout.js (http://localhost:3000/api/knowice/datagen), la función almacena el cuerpo del mensaje, el cual contiene la información del nuevo dispositivo que se da de alta.

Para ello, se crea una variable de tipo *let* llamada *New_data*, y se le asigna el contenido de la petición, almacenada en *req.body.* Las variables tipo *let* son empleadas en este tipo de procesamiento de peticiones, al ser variables locales que existen dentro del bloque de funciones donde se crearon. Adicionalmente, estas variables se eliminan una vez el bloque de funciones se finaliza, liberando memoria en el proceso.

Tras almacenar el contenido de la petición *post,* se realiza un proceso de validación para asegurar que la petición recibida contiene la información necesaria para su correcto procesamiento. Esto se hace comprobando si el cuerpo recibido tiene los atributos requeridos, así como si estos no están vacíos. En cuyo caso se remite un mensaje de error, el cual ha sido procesado para poder enviarse en formato JSON mediante la función *JSON.stringify()*.

Un protocolo de validación es imprescindible a la hora de evitar que la función devuelva errores al tratar de procesar información que no esté presente dentro de parámetros de trabajo. La implementación de dicho protocolo puede verse en el siguiente código.

```
if(New_data.KI_ID != null && New_data.id != null && New_data.empresa != null
  && New_data.location != null && New_data.model != null && New_data.main_capacity != null)
  {
      .
      .
      .
  } else
{
  res.write(JSON.stringify(`Información incorrecta o incompleta`));
  return res.status(400).end();
}
```

*Código 9: Protocolo de validación, routerGEN.post, GenRout.js*

Por lo tanto, si se trata de dar de alta un dispositivo sin aportar la información necesaria, el servidor nos devuelve el código de error 400 y el siguiente mensaje:

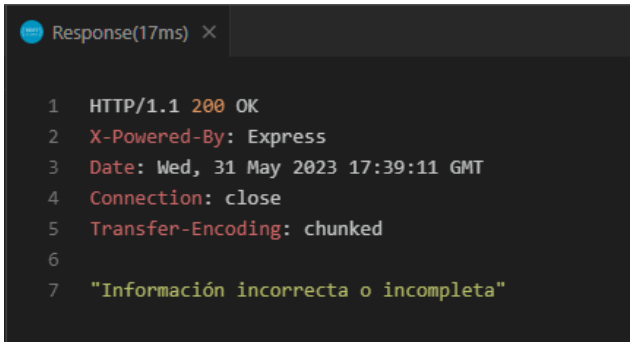



*figura 23: mensaje de error, petición de alta de dispositivo incompleta, routerGEN.post, GenRout.js*

Debe añadirse que la extensión de *REST Client,* en la versión empleada en el presente proyecto, no representa correctamente el código HTTP adjunto con la respuesta, mostrando un código 200 de respuesta satisfactoria siempre.

En el caso de poseer la información requerida, la función procede a comprobar si se trata de un dispositivo cuya ID ya esté presente dentro de la base de datos, haciendo uso de la función *findIndex()*.

Para ello se crea una variable, en este caso llamada *indice*, y se le asigna el resultado de la siguiente línea de código

```
dataGen.findIndex(nevera => nevera.id == New_data.id);
```

*Código 10: findIndex, , routerGEN.post, GenRout.js*

Esta función, al ser aplicada a *dataGen*, busca dentro de esta variable el índice correspondiente al objeto que posea una propiedad llamada "id" con el mismo valor del atributo "id" perteneciente al cuerpo de la petición recibida.

Siendo la base de datos, en esencia, un array que contiene distintos objetos, el valor devuelto por esta función hace referencia a la posición del objeto resultado de la búsqueda dentro de la variable dataGen. Por otro lado, de no dar resultado la búsqueda, la función *findIndex()* nos devuelve el valor de '-1'.

Por lo tanto, si el valor devuelto por la función fuera distinto de -1, significa que un dispositivo con el id indicado en la petición ya figura en la base de datos de la API. En dicho caso, el servidor devolverá el siguiente mensaje de error:

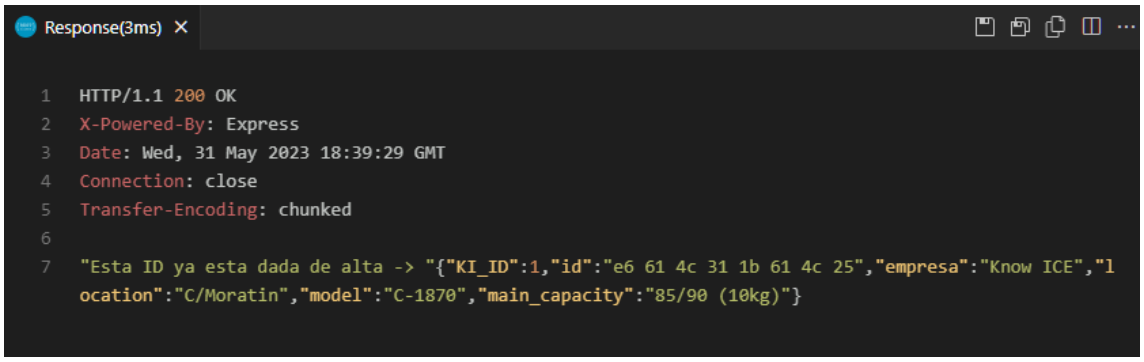



*figura 24: mensaje de error, petición de alta de dispositivo repetido, routerGEN.post, GenRout.js*

En el caso de que el dispositivo no esté dado de alta, la función procede a calcular un número para referenciar el ID de Know Ice S.L., siendo este un valor que se emplea para identificar con mayor facilidad los dispositivos dentro de la base de datos.

Para llevar a cabo este cálculo, la función comprueba los identificadores existentes dentro de la base de datos, en búsqueda del más alto entre ellos. Una vez obtenido el ID más alto existente, se le atribuye el siguiente valor al dispositivo que se da de alta.

Una vez calculado este valor, se introduce el objeto con la información respectiva al nuevo equipo en la variable de dataGen, incluyéndolo al final del vector. Para ello se emplea la siguiente función:

```
dataGen.push(New_data);
```

*Código 11: inclusión de nuevos valores a una variable tipo vector , routerGEN.post, GenRout.js*

Por último, la función cierra la línea de comunicación con el cliente, mediante la emisión de un código de validación de valor 200, y el contenido de la base de datos.

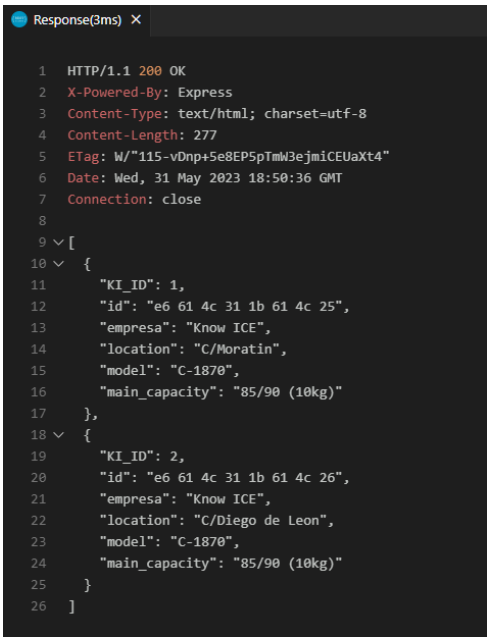



*figura 25: mensaje de respuesta, alta de dispositivoefectuada, routerGEN.post, GenRout.js*

### 3.1.2.5: modificación de dispositivos existentes

Otra función relevante que se debe definir es la modificación de información en la base de datos.

Con el fin de modificar los datos de un dispositivo en concreto, se emplea el protocolo de comunicación HTTP *put*. Este permite la emisión de información parcial sobre un dispositivo en concreto con el fin de actualizar uno o varios valores, sin la necesidad de enviar el objeto completo.

El código empleado para realizar dichas modificaciones es el siguiente:

```
//PUT para modificar un dispositivo existente
routerGEN.put('/', (req,res) => {
  let New_data = req.body;
  if(New_data.KI_ID == null){ //KI_ID como atirbuto imprescindible
      return res.status(400).send(JSON.stringify(`Por favor, indique el ID de Know Ice`));
    }else{
      let indice = dataGen.findIndex(nevera => nevera.KI_ID == New_data.KI_ID);
      if(indice<0){ //no está en la base de datos este disp
        return res.status(404).send(JSON.stringify(`Esta ID de Know Ice no esta dada de alta`));
        }else{ //nos aseguramos de que el resto de los atributos aportados se puedan procesar
          let gen_atri = Object.keys(dataGen[indice]);
          let new_atri = Object.keys(New_data);
          for(i=0; i<new_atri.length; i++){ //
            let ind = gen_atri.findIndex(atributo => atributo == new_atri[i]);
            if (ind < 0){
              return res.status(400).send(JSON.stringify(`la información aportada es
incorrecta`));
            }
          }//si llegó hasta aquí, significa que el paquete de datos está sano
          for(i=0; i<new_atri.length; i++){  //actualizamos la base de datos
              dataGen[indice][new_atri[i]] = New_data[new_atri[i]];
          }
            res.status(200).send(JSON.stringify(dataGen[indice]));
      }
    }
});
```

*Código 12: Procesamiento de peticiones PUT, GenRout.js*

En el ejemplo mostrado en el código 12, nuestra API procesa una petición de tipo *put* recibida a la ruta raíz de GenRout.js (http://localhost:3000/api/knowice/datagen).

En este caso, al tratarse de una modificación de un elemento ya existente en la base de datos, se debe tener en cuenta que no es estrictamente necesario aportar toda la información descrita por el dispositivo. Es suficiente con aportar una propiedad descriptiva y la información que se desea modificar.

Para discernir el dispositivo que se desea modificar, un factor descriptivo como el de KI_ID, es suficiente. Otra opción válida podría tratarse del id único que el microcontrolador de la unidad de venta aporta al ser dado de alta, pero el KI_ID es un identificador más accesible.

Por este motivo, las primeras líneas del código consisten en la identificación del dispositivo que se desea modificar en la base de datos. Para ello, se almacena la información provista por el cuerpo de la petición HTTP y se realiza una verificación en torno a la presencia del identificador *KI_ID*, mediante el siguiente condicional:

```
if(New_data.KI_ID == null){
     return res.status(400).send(JSON.stringify(`Por favor, indique el ID de Know Ice`));
   }else{
                .
                .
                .
```

*Código 13: verificación de identificador, routerGEN.put, GenRout.js*

En el caso de no haberse proporcionado el parámetro en el cuerpo de la petición, la API devolverá una respuesta con el mensaje "Por favor, indique el ID de Know Ice", junto al código 400 (*Bad request*), el cual indica que la petición no pudo ser procesada por el servidor debido a la arquitectura de la propia petición.

Si se posee el identificador, se procede a localizar en la base de datos el dispositivo a modificar, haciendo uso de la función *dataGen.findIndex*(…), almacenando el resultado en la variable *indice* de tipo *let*, tal y como se hizo en el caso de la petición *post*.

Sin embargo, en este caso al detectarse el valor -1, significa que no existe el dispositivo referenciado por la petición en la base de datos, devolviendo el siguiente mensaje "Esta ID de Know Ice no está dada de alta", junto con el código 404 (*Not Found*).

Una vez localizado el dispositivo, se procede a identificar el tipo de información que se desea modificar en la base de datos. Para ello, se analizan los atributos existentes en el cuerpo de la petición.

En el caso del alta de dispositivos (protocolo POST), el análisis del contenido se hace mediante la verificación de que la petición cuenta con los atributos mínimos y necesarios, pudiendo aportarse información adicional a los parámetros básicos.

Sin embargo, en el caso de la petición de tipo *put*, la información que se modifica debe encontrarse entre la existente dentro del propio objeto.

Por lo tanto, para validar la petición, se extraen los atributos aportados por esta mediante la función *Object.keys*(). Esta función devuelve en un vector el nombre de los atributos que pose. Por ejemplo, de querer modificarse la propiedad "*Ubicación*": "*Alberto Aguilera*", la función *Object.keys*() devuelve en un vector el nombre de "*Ubicación*".

Se hace por lo tanto uso del siguiente código para validar la información recibida:

```
let gen_atri = Object.keys(dataGen[indice]);
let new_atri = Object.keys(New_data);
for(i=0; i<new_atri.length; i++){ //
        let ind = gen_atri.findIndex(atributo => atributo == new_atri[i]);
        if (ind < 0){
return res.status(400).send(JSON.stringify(`la información aportada es incorrecta`));
            }
          }
```

*Código 14: verificación de atributos en petición PUT, routerGEN.put, GenRout.js*

Como se puede comprobar, en el código 14, se almacenan los atributos tanto del objeto recibido mediante la petición HTTP, como los atributos del objeto dado de alta en la base de datos. Una vez se tienen estos vectores se comprueba, mediante un bucle *for* y una vez más haciendo uso de la función *findIndex*(…), si los atributos descritos en la petición pueden encontrarse entre los que posee la base de datos.

De localizarse un atributo que no exista entre los existentes en el dispositivo del servidor, inmediatamente se detiene el proceso, devolviendo el siguiente mensaje: "la información aportada es incorrecta", junto con el código de error 400 (*Bad Request*).

En el caso de que la petición pase los filtros antes mencionados, la función procede a actualizar la base de datos con la información recibida, mediante el siguiente código:

```
for(i=0; i<new_atri.length; i++){  //actualizamos la base de datos
        dataGen[indice][new_atri[i]] = New_data[new_atri[i]];
          }
            res.status(200).send(JSON.stringify(dataGen[indice]));
```

*Código 15: actualización de la base de datos routerGEN.put, GenRout.js*

Por último, se envía la base de datos actualizada, junto con el código de verificación 200.

A continuación, se muestra un ejemplo de una petición PUT y su resultado.

```
Send Request
PUT http://localhost:3000/api/knowice/datagen HTTP/1.1
Content-Type: application/json

{
    "KI_ID": "1",
    "empresa": "Empresarios Agrupados",
    "location": "C/Tutor"
}
```

*figura 26: contenido emitido para modificación de la base de datos*

Response(5ms)

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 145
ETag: W/"91-GCiooeTC5oOIWjaiUOIQa7rqUGw"
Date: Fri, 02 Jun 2023 11:03:24 GMT
Connection: close

[
  {
    "KI_ID": 1,
    "id": "e6 61 4c 31 1b 61 4c 27",
    "empresa": "Empresa1",
    "location": "C/Alberto Aguilera",
    "model": "C-1870",
    "main_capacity": "85/90 (10kg)"
  }
]
```

Response(3ms)

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 147
ETag: W/"93-k4QADdbgkpxvgUthqiplj2hx2Rc"
Date: Fri, 02 Jun 2023 11:05:42 GMT
Connection: close

{
  "KI_ID": "1",
  "id": "e6 61 4c 31 1b 61 4c 27",
  "empresa": "Empresarios Agrupados",
  "location": "C/Tutor",
  "model": "C-1870",
  "main_capacity": "85/90 (10kg)"
}
```

*figura 27: contenido de la BdD tras el alta de un equipo (izq) y la respuesta tras la modificación de este (der)*

### 3.1.2.6: Eliminación de dispositivos existentes

Por último, se integra la funcionalidad dedicada a la eliminación de entradas en la base de datos.

Para ello se hace uso del protocolo HTTP *delete*. Esta función es bastante sencilla de aplicar, siendo integrada con el siguiente código en la API:

```
//DELETE para eliminar un dispositivo conociendo su ID (se expresa en la URL)
routerGEN.delete('/:ki_id', (req,res) => {
  const id = req.params.ki_id;
  const indice = dataGen.findIndex(disp => disp.KI_ID == id);
  if(indice >= 0){
    dataGen.splice(indice, 1); //dónde cortamos el array y cuántos borramos a partir de ahí
  }
  else{
    res.write(JSON.stringify(`no se encontro el dispositivo con identificador ${id}`));
    return res.status(404).end();
  }
  res.status(200).send(JSON.stringify(dataGen));
});
```

*Código 16: protocolo de eliminación de paquetes de datos routerGEN.put, GenRout.js*

A diferencia de las anteriores peticiones, el recurso *delete* no aporta información relevante en el cuerpo de la petición. Sin embargo, este se envía no a la URL raíz del router, sino a una dirección terminada con la referencia del dispositivo que se quiere eliminar.

Para eliminar el dispositivo una vez recibida la petición, primero de todo, se localiza en la base de datos el objeto que cuenta con el identificador indicado, haciendo uso de la función *dataGen.findIndex*(…).

De no localizarse el dispositivo referenciado en la URL de la petición, la función procede a enviar el código HTTP 404 (Not Found), acompañado del siguiente mensaje: "no se encontro el dispositivo con identificador (…)", añadiendo el valor de KI_ID indicado por el cliente.

Una vez obtenido el índice, el cual indica la posición del objeto dentro del vector dataGen, se procede a eliminarlo. Esto se hace usando la función *splice()*. Esta función recibe dos atributos, el índice donde "cortará" el vector, y el número de elementos a eliminar. Por lo tanto, se remite en la función el valor de la variable *índice*, y un *1* para el siguiente atributo.

Por último, se remite al cliente el código HTTP 200, indicando que su petición ha sido procesada con éxito, opcionalmente enviándole la nueva base de datos, ahora con el dispositivo eliminado.

### 3.1.3: NewRout.js

El router NewRout.js es un router sencillo que se emplea para el alta de los dispositivos.

Su función consiste en crear el primer puente de comunicación con el dispositivo de venta una vez sea conectado. Esto lo hace poniendo a la disposición del microcontrolador de la cámara frigorífica, a la vez que del cliente, de una ruta URL en la que poder comunicarse.

Una vez que el microcontrolador se energiza por primera vez y establece conexión a la red local del dispositivo de venta, éste tratará de localizar su información en la base de datos, realizando una petición *get* a la dirección http://localhost:3000/api/knowice/datagen/all. para ello emplea su "*id*", el cual es único y específico de cada microcontrolador.

En el caso de no poder encontrarse, lo que haría a continuación será publicar el mismo número de identificación a la siguiente dirección http://localhost:3000/api/knowice/ID/new.

Esta dirección hace la función de la primera interfaz con nuestro dispositivo de venta, redirigiendo las peticiones recibidas por nuestra API al router NewRout.js.

Mediante la búsqueda de la URL antes descrita, se obtiene el siguiente resultado:

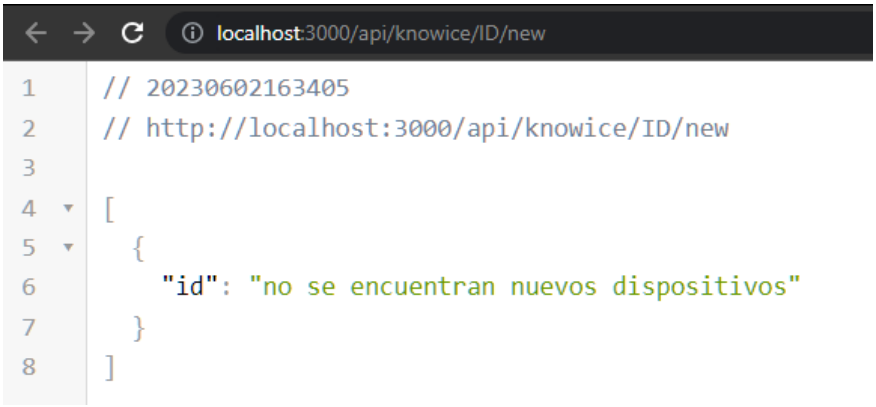



*figura 28: resultado búsqueda http://localhost:3000/api/knowice/ID/new, previo a la conexión del dispositivo*

El resultado de la búsqueda consiste en un objeto en formato JSON, enviado por la API, consistente en un solo atributo "id" con la frase "no se encuentran nuevos dispositivos". Con la puesta en marcha de una máquina frigorífica que no ha sido de alta previamente, esta publica en la misma dirección su identificador, haciendo uso del protocolo HTTP *put*.

El código del router NewRout.js que hace esto posible es el siguiente:

```
const express = require('express');
const routerNEW = express.Router();

const newID = [
  {id: 'no se encuentran nuevos dispositivos'}
]



routerNEW.get('/', (req,res) => { //en esta ruta se pueden ver todos los disp
    res.send(newID);
});

routerNEW.use(express.json()); //IMPORTANTE!!!!!! esto va a permitir procesar el cuerpo de la
solicitud en formato JSON (middleware)
//el middleware se ejecuta después de recibir una solicitud y antes de envuiar una respuesta

//aquí es donde la raspberry que no esté dada de alta pondrá su id
routerNEW.put('/', (req,res) => {
  let New_ID = req.body;
  newID[0] = New_ID;
  res.statusCode=200;
  res.send(JSON.stringify(newID));
});


module.exports.routerNEW = routerNEW;
```

*Código 17: NewRout.js*

Consultando esta dirección, el usuario es capaz de extraer el identificador del dispositivo de control de la cámara frigorífica, y darlo de alta en la base de datos. Una vez hecho esto, el propio microcontrolador extrae la información necesaria para su funcionamiento de la base de datos y elimina su identificador de la URL vinculada a NewRout.js, dejando de nuevo la frase "no se encuentran nuevos dispositivos".

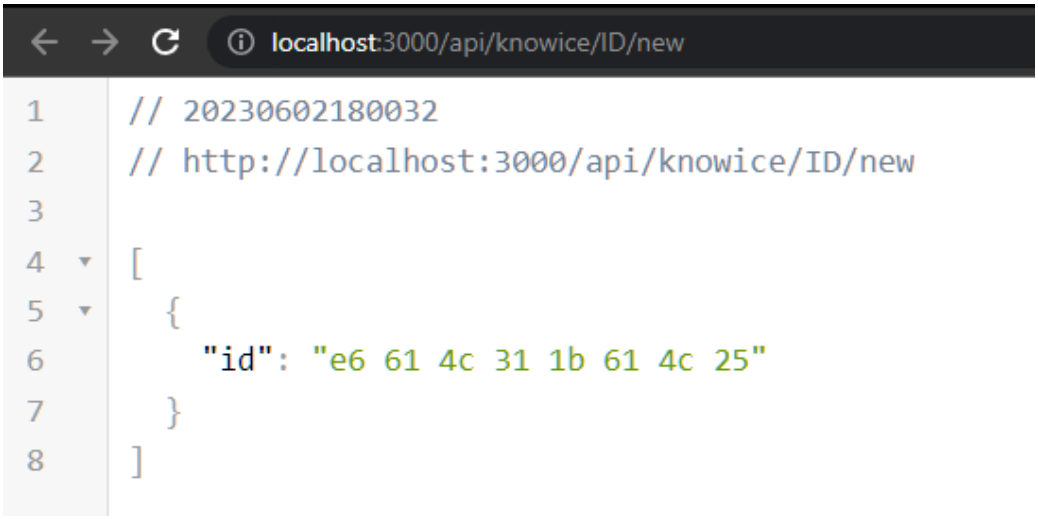



*figura 29: resultado búsqueda http://localhost:3000/api/knowice/ID/new, posterior a la conexión del dispositivo*

## 3.2 Raspberry Pi Pico W

El microcontrolador empleado para el control de la cámara frigorífica se trata de una Raspberry Pi Pico W, esta centraliza las instrucciones a seguir por los actuadores y sensores del dispositivo (finales de carrera y actuadores de apertura de puertas, sensores de proximidad y temperatura dentro de la cámara…), al mismo tiempo que coordina el funcionamiento del terminal de pago y registra el estatus del dispositivo y su historial de ventas.

La Raspberry Pi Pico emplea el lenguaje de programación de micropython, empleando la plataforma de programación Thonny, somos capaces de inyectar en el microcontrolador las librerías de micropython y programar directamente sobre el dispositivo.

El lenguaje de Micropython consiste en una versión simplificada del lenguaje de programación Python 3, orientado a microcontroladores y optimizado para poder funcionar en dispositivos ocupando hasta 256kb de memoria y 16kb de RAM.

Thonny, a diferencia de otras plataformas de programación de microcontroladores, no realiza la función de *debugging*, sino que lo hace directamente la Raspberry. Es decir, se emplea la Raspberry para analizar y ejecutar el programa, el cual se almacena en su interior y es procesado por las librerías específicas de micropython que le han sido inyectadas.

Esta supone una gran diferencia con respecto a otras plataformas de programación, como la de Arduino, permitiendo internalizar el proceso de *debugging* en el dispositivo de control, sin comprometer significativamente por ello la memoria del dispositivo. De esta manera es posible emplear la Raspberry como una especie de "PenDrive", pudiendo acceder a los programas que se ejecutan en su interior, con tan solo conectarla al ordenador.

El trabajo de programación que se ha realizado en el dispositivo de control se puede dividir en los archivos que se han generado para el correcto funcionamiento de este.

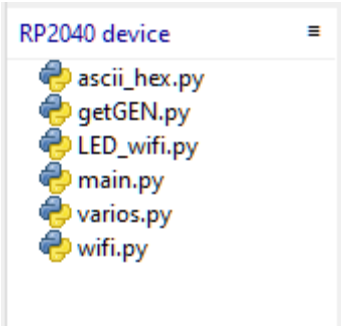



*figura 30: documentos PY*

El primer fichero a comentar es el de "main.py", este centraliza las funciones de control y es el programa que se ejecuta una vez se enciende la Raspberry.

El siguiente fichero relevante es "wifi.py", este inicializa las funciones de comunicación WiFi del dispositivo, conectándose a la red privada descrita en su código y con la contraseña provista.

En paralelo a "wifi.py", se ejecuta LED_wifi.py, la función de este archivo es la de mostrar, mediante el indicador LED construido en la propia placa, el estado actual de la conexión WiFi.

El siguiente archivo a mencionar es el de "getGEN.py", este posee las funciones necesarias para consultar a la base de datos de Know Ice S.L. los valores de funcionamiento del dispositivo, al mismo tiempo se encarga de transmitir el identificador único del dispositivo si este fuera necesario para dar de alta el dispositivo.

Con relación al anterior archivo, se encuentra “ascii_hex.py”. Este es un archivo que cuenta con la función de modificar el formato del identificador único del dispositivo Raspberry, de tal manera que este no se altere al poseer caracteres que no son capaces de procesar el lenguaje HTML.

Por último, el archivo “varios.py” realiza las funciones de usos varios que serán empleadas por el resto de los archivos del programa.

A continuación, se explica el funcionamiento de las distintas partes del programa de control en la fase final de la realización del proyecto.

### 3.2.1: main.py

Ante la inicialización de la Raspberry, el microcontrolador ejecuta por defecto el programa en su interior que corresponda al nombre de “main.py”, es por este motivo que el programa principal se estructura en este archivo.

La programación del microcontrolador, a diferencia de la que se ha representado hasta ahora de Javascript, consiste en una programación unifilar síncrona. Es decir, mientras la plataforma de Node.js ejecuta distintas líneas de código, de manera simultánea sin seguir una estructura lineal, el microcontrolador de Raspberry ejecuta el código de manera ordenada y lineal, sin la capacidad de ejecutar distintas funciones de manera simultánea.

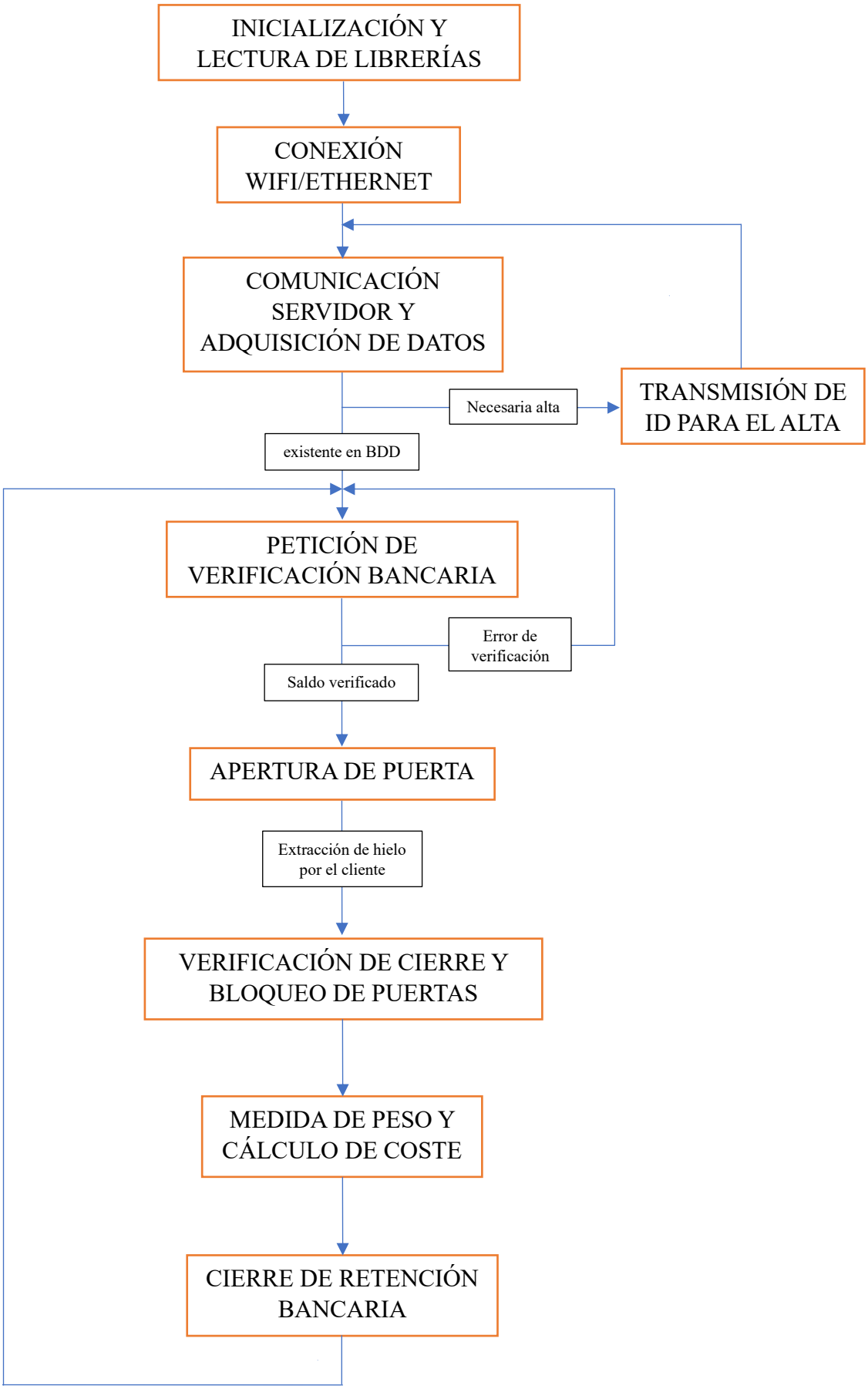
Por otro lado, el microcontrolador de Raspberry Pi Pico W cuenta con dos núcleos que son capaces de ejecutar “hilo” de código, es decir, ejecuta el programa de forma ordenada, no pudiendo realizar acciones en paralelo las unas de las otras. Cada uno de estos núcleos posee acceso a una memoria dedicada, pudiendo compartir información entre ellos mediante un protocolo FIFO (First In First Out) a través de una memoria común.

Aun siendo verdad que el microcontrolador posee dos núcleos integrados, significando que podría completar las acciones de dos códigos distintos en paralelo, esta función está integrada de tal forma que su aplicación a menudo debilita la robustez del programa, pudiendo entorpecer o bloquear el funcionamiento de control principal debido a un error, requiriendo un reinicio forzoso para superar el bloqueo.

Esta precariedad se hace de notar especialmente ante la implementación de protocolos web HTTP. Esto se debe al uso común por parte de los dos núcleos del mismo “reloj” u oscilador, pudiendo generar un conflicto entre los dos núcleos, impidiendo que se complete la petición HTTP satisfactoriamente.

Es por ello por lo que se sugiere el empleo de los dos núcleos únicamente para funciones sencillas o metódicas, evitándolo en la mayor medida posible ante procesos complejos como pueden ser los de comunicación web.

Por lo tanto, para realizar la programación del microcontrolador, se ha de tener en cuenta la secuencia de acciones que deberá ejecutar durante el funcionamiento en operación, en la siguiente página se representa en un diagrama de bloques la arquitectura principal sobre la que se regirá el programa.


INICIALIZACIÓN Y LECTURA DE LIBRERÍAS
CONEXIÓN WIFI/ETHERNET
COMUNICACIÓN SERVIDOR Y ADQUISICIÓN DE DATOS
Necesaria alta
TRANSMISIÓN DE ID PARA EL ALTA
existente en BDD
PETICIÓN DE VERIFICACIÓN BANCARIA
Error de verificación
Saldo verificado
APERTURA DE PUERTA
Extracción de hielo por el cliente
VERIFICACIÓN DE CIERRE Y BLOQUEO DE PUERTAS
MEDIDA DE PESO Y CÁLCULO DE COSTE
CIERRE DE RETENCIÓN BANCARIA

Por desgracia, debido al abandono del proyecto por parte de la empresa Know Ice S.L., junto con la dimisión del tutor de este proyecto de dicha empresa y por consecuente, su pérdida de compromiso para con esta, el presente proyecto fue puesto en *stand-by* indefinidamente.

Por lo tanto, ante la falta de un hardware dedicado a la medida precisa de las galgas de peso, con la cualificación necesaria para operar en materia de producto orientado a la venta de bienes por peso, no se pudo teorizar el funcionamiento de comunicación Raspberry / conversor analógico digital. Haciendo de la finalización del programa "main.py" un acto impracticable.

Por este motivo el programa "main.py" permanece en el estado inacabado que se puede observar en la siguiente imagen.

```
import wifi
import LED_wifi
import getGEN
import varios

import urequests as requests
import ujson
import time


LED_wifi.init()
time.sleep(1)
wifi.init()


#while True:
while(LED_wifi.LAN_status == False): #nos quedamos aquí hasta que esté conectado
   time.sleep(0.1) #puedo importar variables!!!! Time.sleep mejor que pass

GEN = getGEN.get_general_data()
```

*Código 18: main.py*

Como se puede apreciar en el Código 18 recién mostrado, las primeras cinco líneas de código corresponden a las importaciones del resto de ficheros incluidos en la memoria de la Raspberry, esto permitirá hacer uso de las funciones y consultar el valor de las variables que estos posean más adelante.

Las líneas de código subsecuentes realizan la inclusión de la librería "*urequests*", a la que se referirá en el resto del código como "*requests*", y "*ujson*".

La primera de estas, "*urequests*", se trata de una librería empleada para la comunicación mediante protocolo web con otros dispositivos. En el caso de este proyecto, es empleada para la emisión y recepción de solicitudes HTTP con el servidor de Know Ice S.L., y la comunicación del dispositivo de control con el terminal de pago de Paytef asociado a la máquina de venta.

La siguiente librería, "*ujson*", facilita el procesamiento e interpretación de los paquetes de datos empleados en la comunicación HTTP, transformándolos de formato texto a JSON y viceversa.

Por último, se incluye la librería "time", esta permite hacer operaciones basadas en el tiempo, más concretamente se emplea con el objetivo de realizar pausas de un tiempo determinado dentro del propio código.

Posteriormente a la iniciación del programa, se encuentra la llamada de la función "*init*()", perteneciente al programa "*LED_wifi.py*". Esta inicia la función de interrupción encargada de señalizar en el LED integrado en la placa, el estado actual de la conexión WiFi del dispositivo.

Una vez iniciada la función de *LED_wifi*, se procede a establecer la conexión del dispositivo con la red local, esto se hace llamando a la función "*init*()" del dichero "*wifi.py*".

A continuación, se realiza un bucle en el que permanece a la espera de que se establezca la conexión a la red, haciendo uso para ello la variable "*LAN_status*", perteneciente al archivo "*LED_wifi.py*" como condicional.

Por último, se realiza la llamada a la función "*get_general_data*" perteneciente al fichero "*getGEN.py*", y depositando el valor devuelto en la variable creada como "GEN".

### 3.2.2: wifi.py

A continuación, se muestra el contenido del documento "*wifi.py*":

```
import LED_wifi
import network
import time
from machine import Timer

timCheck = Timer()

def connect(): #esta función conecta con el wifi, si ya está conectado devuelve True
    #print("connecting to network...")
    sta_if = network.WLAN(network.STA_IF)
    if not sta_if.isconnected():
        sta_if.active(True)
        sta_if.connect("[redacted]", "[redacted]")
        time.sleep(1)
    else:
        pass
        #print("connected")
        #print("network config", sta_if.ifconfig())
    return sta_if.isconnected()

def init(): #esta función se llama una sóla vez, inicializa el temporizador que llama a la función de conectar
    connect()
    timCheck.init(freq = 0.5, mode = Timer.PERIODIC, callback = check)

def check(timer): #timer no sé qué es, pero sin algo escrito aquí no funciona
    #print("check")
    LED_wifi.status(connect())
```

*Código 19: wifi.py*

El programa "*wifi.py*" contiene las funciones necesarias para la activación de la conexión wifi, estableciendo conexión con la red de nombre y contraseña especificados en el propio código.

Al igual que en el caso del programa "*main.py*", las primeras líneas de código de "*wifi.py*" consisten en la importación de las librerías y archivos que se emplean en el resto del documento; en este caso, la primera de ellas se trata del programa "*LED_wifi.py*", esto se debe a que la comunicación entre ambos ficheros se hace a través de la función status() de "*LED_wifi.py*".

La siguiente que se incorpora se trata de la librería "*network*". Esta habilita la posibilidad de llamar a funciones relacionadas con el módulo wifi integrado en el microcontrolador, y será la que se emplee para establecer y comprobar la conexión más adelante.

El último elemento que se integra y debe comentarse, es el del "*Timer*". Esta es una función perteneciente a la librería "*machine*", que habilita la posibilidad de activar temporizadores. De esta manera, permite especificar al reloj integrado en el microcontrolador un margen de tiempo, en el cual debe realizar una función determinada. Dicha acción se basa en interrumpir el programa principal, que es reanudado cuando la función realizada por el *timer* finaliza.

Para hacer esto posible, primero ha de generarse una variable de tipo *timer.* En el caso de este código, dicha variable se llama "*timCheck*", y se le asocia la función "*Timer*()" tal y como se puede ver en la línea 6 del código 19.

#### *3.2.2.1: init()*

La activación del temporizador se realiza desde la función definida como "*init*()". Esta función es llamada por primera vez durante el proceso de inicialización del microcontrolador, en el propio programa de "*main.py*", tal y como se puede ver en la línea 13 perteneciente al código 18.

La ejecución de la función "*init*()" consiste en la llamada de la función "*connect*()", definida más arriba que esta, para evitar la llamada de una función que no haya podido ser "leída" con anterioridad.

Tras la ejecución de la función "*connect*()", cuyo papel se describe en detalle más adelante, se inicia el temporizador mediante la siguiente función:

```
timCheck.init(freq = 0.5, mode = Timer.PERIODIC, callback = check)
```

*Código 20: inicialización del temporizador, wifi.py*

Con la estructura mostrada en el código 20, se activa la variable "*timCheck",* la cual ha sido asociada a la función "*Timer*()". A este temporizador se le atribuyen las siguientes características: "freq" indica la frecuencia, en Hercios, con la cual será llamada la función asociada al temporizador. En este caso el valor de "*freq*" es de 0.5Hz, significando que la función va a ser llamada cada 2 segundos, interrumpiendo el programa principal en el proceso.

Alternativamente al parámetro "*freq*", existe la posibilidad de emplear el parámetro "*period*". Este indica el tiempo entre la finalización de la función asociada al *timer* y la siguiente llamada a esta, dicho valor se expresa en milisegundos.

El siguiente atributo se trata de "mode". Este indica el tipo de procedimiento en el cual se ejecuta la función timer. En el caso de "*wifi.py*", se trata de una función periódica (PERIODIC). Esto quiere decir, que la función asociada al *timer* se ejecuta una vez por cada tiempo especificado en el parámetro anterior.

Además de la función *PERIODIC*, existe el parámetro *ONE_SHOT*. Su forma de proceder consiste en ejecutar la función asociada durante el tiempo indicado en el atributo anterior. La función asociada se ejecutará por el temporizador una única vez y cada vez que dicho temporizador sea inicializado.

Por último, el parámetro "*callback*" se emplea para definir la función que se asocia al temporizador, en este caso se trata de la función "*check*()".

#### *3.2.2.2: check()*

Siempre y cuando la función "*check*()" funcione en coordinación al temporizador "*timCheck*", esta debe definirse con el parámetro "*timer*" como requisito en los parámetros de llamada a la función, tal y como puede verse en el código 21.

```
def check(timer): #timer no sé qué es pero sin algo escrito aquí no funciona
  #print("check")
  LED_wifi.status(connect())
```

*Código 21: definición de la función check(), wifi.py*

La función "*check*()" consiste en la llamada de la función "*status*()", perteneciente al código "*LED_wifi*". Para la ejecución de dicha función es necesario aportar en los parámetros de llamada un valor de tipo booleano, es decir, un Verdadero o Falso.

Dicho valor se obtiene como resultado de la llamada a la función "*connect*()", perteneciente al propio código "*wifi.py*". La nomenclatura puede reducirse de tal forma que la llamada de la función "*connect*()" se incorpore dentro de los parámetros de llamada de la función "*status*()".

Por lo tanto, cada vez que el temporizador se active (es decir, cada 2 segundos de tiempo real de ejecución del código principal), la función " *check*()" se ejecutará, haciendo llamar a la función "*connect*()" y remitiendo la respuesta de este a la función "*status*()".

#### *3.2.2.3: connect()*

La función definida como "*connect*()" al inicio del código, hace una doble función de conexión a la red y verificación del estado de dicha conexión.

La primera parte de la función define una variable, denominada "*sta_if*", con las propiedades de la librería "*network*" referentes a la conexión *Wireless*. Una vez definida dicha variable, esta tiene las propiedades propias de una variable de clase *"network"*.

La primera función empleada de esta clase es la descrita como ".isconnected()". Esta función devuelve el estado de la red llamada "*sta_if*", respondiendo el valor *True* en caso de que la conexión haya sido realizada en esta red, o *False* en caso contrario.

```
def connect(): #esta función conecta con el wifi, si ya está conectado devuelve True
  #print("connecting to network...")
  sta_if = network.WLAN(network.STA_IF)
  if not sta_if.isconnected():
    sta_if.active(True)
    #sta_if.connect("El Felimóvil", "zatn7026")
    sta_if.connect("[redacted]", "[redacted]")
    time.sleep(1)
  else:
    pass
    #print("connected")
    #print("network config", sta_if.ifconfig())
  return sta_if.isconnected()
```

*Código 22: función connect(), wifi.py*

Haciendo uso del estado de la red a través de un condicional, por lo tanto, se discierne entre las acciones necesarias a tomar. De este modo se acorta la ejecución de la función cuando la conexión WLAN está activa.

En el caso de que el resultado de la función *sta_if.isconnected()* de positivo, se remite directamente al final de la función, en el cual se realiza de nuevo la llamada a la función *sta_if.isconnected(),* devolviendo directamente el valor resultado al código desde donde fue llamada la función. En este caso, la función "*check*()".

En el caso de que *sta_if.isconnected()* devuelva un valor negativo, se procede a la activación de las funciones WiFi del microcontrolador, mediante la función *sta_if.active(True)*. Una vez activado el módulo WiFi de la placa microcontroladora, se le solicita al módulo la conexión a la red local de nombre y contraseña definida por los parámetros incluidos en la función *sta_if.connect*(<nombre de red>,<contraseña>).

Este proceso se realiza cada dos segundos, siempre que la conexión red se encuentre inactiva, activando el módulo y definiendo el *ssid* y la contraseña de la red. Esto, sin embargo, no entorpece a la conexión, la cual tarda en realizarse entre cuatro y cinco ciclos de la función periódica "*connect*()".

El motivo por el cual se realiza esta función de forma periódica consiste en la necesidad, o propiedad, de nuestro microcontrolador de comunicar el estado de la red mediante el LED instalado en la propia placa. De esta forma se facilita en gran parte la diagnosis de un mal funcionamiento, descartando un error de conexión (si así lo indica el LED) o un bloqueo total del programa en caso de que este no conmute pasado un tiempo.

### 3.2.3: LED_wifi.py

El siguiente programa, "*LED_wifi.py*", realiza la función de señalización del estado de conexión red del módulo wifi incorporado en la Raspberry Pi Pico W.

A continuación, se muestra el código completo de "LED_wifi.py" (código 23)

```
from machine import Pin, Timer

tim = Timer()
led = Pin("LED", Pin.OUT)
LED_state = True
LED_freq = 0.4
LAN_status = False
i = 0

def init(): #nos inicia el timer, sólo se llama una vez
  tim.init(freq = LED_freq, mode = Timer.PERIODIC, callback = tick)

def status(wifi): #nos trae la información del estado del wifi
  global LAN_status
  LAN_status = wifi

def tick(timer): #función que controla el LED dependiendo del estado del LAN
  global led, LED_state, LED_freq, LAN_status, i
  if LAN_status == True: #oki
    if LED_state:
      LED_freq = 0.5
    else:
      LED_freq = 20
    LED_state = not LED_state
  else: #not oki
    if i == 0:
      i=1
      LED_freq = 20
      LED_state = True
    elif i == 1:
      i=2
      LED_freq = 10
      LED_state = False
    elif i == 2:
      i=3
      LED_freq = 20
      LED_state = True
    elif i == 3:
      i=0
      LED_freq = 0.5
      LED_state = False

  led.value(LED_state)
  tim.init(freq = LED_freq, mode = Timer.PERIODIC, callback = tick)

#definimos la frecuencia, el modo de timer y a qué tiene que llamar de tim
```

*Código 23: LED_wifi.py*

Para la ejecución del código "*LED_wifi.py*" es necesario la inclusión de las librerías "*Pin*" y "*Timer*", pertenecientes al conjunto "*machine*". La primera de estas, la librería "*Pin"*, permite la activación de las funciones IO y la lectura de los puertos analógicos. En este

caso, el puerto empleado en el presente código se trata de la salida digital al led incorporado en la placa, la cual se referencia como "LED" por la propia librería de funciones.

La Primera acción que se realiza, tras la importación de las librerías, se trata de la creación de un elemento de tipo "*Timer*()", al igual que se vio en el anterior apartado de "*wifi.py*". En este caso el objeto con las propiedades del temporizador se referirá como "*tim*".

La siguiente línea de código se emplea para la declaración del puerto asignado al led incorporado en la placa, como un puerto de salida. Esto se hace mediante la siguiente declaración: *led = Pin("LED", Pin.OUT)*, esto permite modificar el estado del LED mediante la siguiente línea de código: *led.value( ),* indicando en el paréntesis el estado del led deseado mediante un valor booleano.

Las siguientes líneas de código, están dedicadas a la declaración de variables globales, con sus valores iniciales, que serán empleadas por el resto del programa. Estas son: "*LED_state*" como variable booleana con valor inicial *True* para indicar el estado del LED, "*LED_freq*" la cual servirá para la declaración del timer, indicando, en Hercios, el tiempo entre interrupciones, "*LAN_stastus*" para guardar constancia del estado de la red WLAN, e "*i",* empleado como contador de etapas para realizar las funciones de la máquina de estados que se verá más adelante.

#### *3.2.3.1: init()*

La primera función definida dentro del archivo "*LED_wifi.py*" se trata de la función init(). Esta se exporta al código principal "*main.py*" y se emplea para la inicialización del temporizador asignado al led.

```
def init(): #nos inicia el timer, sólo se llama una vez
  tim.init(freq = LED_freq, mode = Timer.PERIODIC, callback = tick)
```

*Código 24: iniciación del temporizador, init(), LED_wifi.py*

Durante el arranque de la Raspberry, el código principal "*main.py*" realiza una llamada a la función "*init*()" mostrada en el código 24. Esta inicia el temporizador, desencadenando las funciones de la máquina de estados encargada de mostrar el estado de la red WLAN.

La función "*init*()" es llamada tan sólo una vez, durante el arranque del programa, e inicia el temporizador con los valores de estado declarados al inicio de "*LED_wifi.py*", estableciendo una llamada de carácter periódica a la función "*tick*()" cada 2,5 segundos (0,4Hz).

#### *3.2.3.2: status()*

La función "*status*()" se trata de un sistema de comunicación establecido para la exportación del estado de la red WLAN. Esta función se exporta al archivo "*wifi.py*", el cual hace la llamada a la función periódicamente, haciendo uso de la función "*check*()" tal y como se puede ver en el código 21.

```
def status(wifi): #nos trae la información del estado del wifi
    global LAN_status
    LAN_status = wifi
```

*Código 25: función status(), LED_wifi.py*

La primera acción realizada en la función "*status*()" es la importación de la variable global "*LAN_status*", la cual fue declarada al inicio de "*LED_wifi.py*", esto se realiza mediante el protocolo "*global*". De esta manera la función puede, no sólo consultar, sino modificar el estado de dicha variable. De no ser por esta acción, las declaraciones realizadas sobre la variable "*LAN_status*" dentro de la misma función, se realizan de forma local, eliminándose una vez han sido terminadas las tareas de la función "*init*()".

La siguiente línea corresponde a la declaración de estado de la variable "*LAN_status* ()", la cual toma el valor del parámetro de llamada de la función en su origen, referido en la propia función como "*wifi*". En este caso, dicho parámetro toma el valor resultante de la función "*connect*()" (código 22), la cual, a su vez, es llamada igualmente por la función "*check*()" (código 21). De esta forma, se permite la exportación del estado de la red WLAN al programa "*LED_wifi.py*".

#### 3.2.3.3: tick()

```
def tick(timer): #función que controla el LED dependiendo del estado del LAN
    global led, LED_state, LED_freq, LAN_status, i
    if LAN_status == True: #oki
        if LED_state:
            LED_freq = 0.5
        else:
            LED_freq = 20
        LED_state = not LED_state
    else: #not oki
        if i == 0:
            i=1
            LED_freq = 20
            LED_state = True
        elif i == 1:
            i=2
            LED_freq = 10
            LED_state = False
        elif i == 2:
            i=3
            LED_freq = 20
            LED_state = True
        elif i == 3:
            i=0
            LED_freq = 0.5
            LED_state = False

    led.value(LED_state)
    tim.init(freq = LED_freq, mode = Timer.PERIODIC, callback = tick)
```

*Código 26: máquina de estados tick(), LED_wifi.py*

La función "tick()" realiza la función de modificar el estado del LED instalado en la placa del microcontrolador. Esto lo hace mediante dos máquinas de estado, una para cada posible estado de conexión de la red WLAN.

Al ser “tick()” una función que es activada periódicamente mediante un temporizador se debe incluir “*timer*” entre los parámetros de llamada de la función. De no ser así, no podría completarse el protocolo de interrupción.

Para el correcto funcionamiento de la máquina de estados, se debe importar el valor de los parámetros a emplear como variables globales. De esta forma el valor de las variables empleadas por la función se mantiene entre el final de la ejecución de la función y el inicio de la siguiente llamada. Si se tratarse de variables locales, estas reinician su estado en cada llamada de la función “*tick*()”.

Para la ejecución del presente código, se importan las variables de “LED_state”, “LED_freq”, “LAN_status”, “i” y “led”, la cual contiene las propiedades del puerto digital de LED necesarias para modificar el estado de este.

La máquina de estados se desdobla en dos modos de funcionamiento, en base al estado de la variable “LAN_status”. El primer condicional de la función realiza la diferenciación entre la correcta conexión WLAN del dispositivo (tomando *LAN_status* el valor *True*) y el error de conexión del módulo (*False*).

En el primer caso del condicional, correspondiente al valor *True* de la variable “*LAN_status*”, el LED del dispositivo realiza un pulso de 0,05 segundos (20 Hercios), seguido de una pausa con el LED apagado de 2 segundos (0.5 Hercios).

En caso de que la red WLAN no esté disponible o activa, (*LAN_status* toma el valor *False*), el LED del dispositivo realiza un ciclo de dos pulsos consecutivos, cada uno de 0,05 segundos (20 Hercios), y separados entre sí por una pausa de 0,1 segundos (10 Hercios). Tras dicho ciclo, el LED del dispositivo permanecerá apagado durante una pausa de 2 segundos (0.5 Hercios).

Para llevar a cabo este ciclo, la función cuenta con una variable global “*i*” empleada para mantener constancia del estado del LED en el ciclo. Esta variable se emplea para discernir de los cuatro posibles estados de la función en condiciones de desconexión de la red.

Las modificaciones del estado del LED se realizan en la penúltima línea del código, atribuyéndole al puerto lógico de la placa el valor de la variable “LED_state”. Los cambios de estado del LED, por lo tanto, se realizarán al final de cada ejecución de la llamada a la función “*tick*()”.

Los tiempos entre llamadas de la función “*tim*” se alteran mediante la modificación de la declaración de dicha función timer, inicializándola al final del ciclo de ejecución con los nuevos parámetros de frecuencia.

### 3.2.4: getGEN.py

El programa comprendido en el archivo "getGEN" contiene las funciones necesarias para realizar la comunicación entre la Raspberry y el servidor de Know Ice S.L. que se simula en el presente proyecto.

Concretamente, se define la función "*get_general_data*()", la cual es exportada y llamada por el programa principal "*main.py*" con el fin de obtener, de la base de datos del servidor, la información necesaria para el correcto funcionamiento del dispositivo de venta.

```
"""esta función busca el ID de la RB en la base de datos"""
"""si no aparece, postea el ID para que se dé de alta"""

import ascii_hex #los caracteres de la ID da problemas al postearlo en el servidor, así lo podemos manejar
import urequests as requests
import machine, time, ujson


ip = "http://192.168.1.11"
direc_gen = ":3000/api/knowice/datagen/all"
direc_new = ":3000/api/knowice/ID/new"
#esto de arriba mejor dejarlo en otro sitio?
URL1= ip + direc_gen
URL2= ip + direc_new

def get_general_data():
  ID = ascii_hex.dump_mem(machine.unique_id()) #dump mem te lo pasa a hexadecimal de ascii
  while True:
    try:
      datagen = requests.get(URL1).json()

      for i in range(len(datagen)):
        if ID == datagen[i]["id"]:
          GEN = datagen[i]
          datagen = 0 #liberamos memoria
          print(GEN)
          return GEN
      #si llegamos aquí es que no estamos
      post_data = ujson.dumps({'id': ID})
      requests.put(URL2, headers = {'content-type': 'application/json'}, data = post_data)
      datagen = requests.get(URL1).json()

      while True: #se quedará aquí esperando a que alguien lo de de alta
        datagen = requests.get(URL1).json()
        for i in range(len(datagen)):
          if ID == datagen[i]["id"]:
            GEN = datagen[i]
            datagen = 0 #liberamos memoria
            post_data = ujson.dumps({'id': 'no se encuentran nuevos dispositivos'})
            requests.put(URL2, headers = {'content-type': 'application/json'}, data = post_data)
            print(GEN)
            return GEN
        time.sleep(1)
    except:
      time.sleep(1)
    else:
      pass

#la ID sólo se postea una vez y la sobreescribe cuando se ha dado de alta
```

*Código 27: getGEN.py*

Para realizar las funciones necesarias, el código "getGEN.py" importa las siguientes librerías: "*urequests*" (la cual pasa a referenciarse como "*requests*") para poder aplicar protocolos de comunicación tipo HTTP. La librería "*machine*" para acceder al identificador único inherente al propio procesador de la Raspberry. La librería "time" para efectuar pausas en el código y por último "*ujson*" con el fin de procesar y formatear las variables a formato JSON. Adicionalmente se importa el archivo "*ascii_hex*", el cual se explica más adelante.

A continuación, se declaran las siguientes variables: "*ip*" haciendo referencia a la dirección raíz del servidor local que empleamos como simulador del servidor de Know Ice S.L. Este se ejecuta en el ordenador, generando un entorno accesible al resto de dispositivos conectados a la red local. Al acceder a esta dirección desde un dispositivo ajeno al ordenador, debe escribirse con la dirección IP local del ordenador (http://192.168.1.11).

La siguiente variable describe la ruta en la que se albergan las funciones de nuestro servidor; en el caso de "*direc_gen*" la dirección se completa como ":3000/api/knowice/datagen/all". A excepción del ":3000", el cual es el puerto de escucha de nuestro servidor, los parámetros de subdominio son los encargados de redirigir las peticiones recibidas por el servidor hacia el router "*GenRout.js*" (véase el apartado 3.1.2). Es en esta ruta donde se alberga la información general de los dispositivos dados de alta en la base de datos.

La siguiente dirección descrita es la de "*direc_new*", esta dirección (":3000/api/knowice/ID/new") redirige las peticiones recibidas por el servidor al router "*NewRout.js*" (véase el apartado 3.1.3). Es en esta ruta donde el microcontrolador puede acceder a una variable, la cual es capaz de sobrescribir con el valor de su ID, para ser consultada por el usuario durante el alta del dispositivo.

La combinación de la *ip* con las direcciones descritas anteriormente, generan dos direcciones *URL* (URL1 y URL2). Estas son las que el programa emplea para la consulta y emisión de peticiones HTTP.

#### *3.2.4.1: Función get_general_data()*

La primera parte de la función "*get_general_data*" se encarga de adquirir la información general de todos los dispositivos dados de alta en la base de datos de Know Ice S.L. Una vez obtenida dicha información trata de localizar en ella el paquete de datos correspondiente a un dispositivo con su mismo ID.

Para ello, primero obtiene el identificador único. La función "*machine.unique_id*" se emplea para consultar al procesador de la Raspberry (RP2040) el número de serie de la memoria flash. Este identificador es único para cada Raspberry, por lo que lo emplearemos como base para que la propia placa microcontroladora sea capaz de localizar su información en la base de datos.

En el caso particular de la placa de Raspberry empleada en la realización de este proyecto devuelve como respuesta a la función "*machine.unique_id*" el siguiente identificador: "b'\xe6aL1\x1baL%' ". Como ejemplo adicional a este, el resultado devuelto por una placa Raspberry Pi Pico W similar es el siguiente: "b'\xe6a\x18\xc4\xe3R@)' ".

Una vez obtenida y procesada la ID (ver punto 3.2.5), la función entra en un bucle de tipo *while*. Esto es para evitar las salidas del código debido a errores en la comunicación con el servidor.

Debido a que la versión de micro Python empleada para el desarrollo del código (v1.20.0 - 26/04/2023 [DAMI23]) se encuentra en parte en fase de pruebas a día de realización del presente proyecto, existen ciertos protocolos y funciones que pueden resultar inestables.

En particular, los protocolos pertenecientes a la librería "*urequests*" tienen la capacidad de desencadenar en errores durante los procesos de transmisión y recepción de paquetes de datos vía protocolo HTTP, pudiendo paralizar o bloquear el funcionamiento del programa.

Para evitar esto, empleamos el protocolo *try/except/else.* Esta es una función propia del lenguaje de micro Python, su función es proteger el programa frente a fallos, evitando errores capaces de detener los procesos que realiza.

Para implementarlo, se usa la función "*Try:*" escribiendo la porción de código que se quiere proteger. Una vez introducido el código principal, a continuación, fuera de "*Try*:", se introduce la función "*Except:*".

Dentro del apartado de *except*, se introducen las líneas de código que deben ejecutarse en el caso de que suceda un error durante la ejecución del código principal. En este caso, ya que se desea que se vuelva a intentar ejecutar el código, se realiza una pausa de un segundo para que el error de comunicación con el servidor "se subsane".

Por último, se introduce la función "Else:" fuera del conjunto de "Except:". En este apartado se deben introducir las líneas de código que se desea que se ejecuten en el caso de que el código principal se ejecutara exitosamente. En este caso, se emplea el comando "*pass*".

El comando "*pass*" sirve para indicar un punto del código donde no se debe realizar ninguna función, ya que las funciones *Try/Except/Pass* requieren de un contenido para no devolver un error en la compilación del programa.

Se hace uso del comando "*pass*", ya que se desea que el código de la función "*get_general_data*s" se ejecute las veces que sean necesarias hasta que el proceso de la toma de datos llegue a su fin.

Sólo cuando el código haya realizado exitosamente su función, llegará al comando "*return*", devolviendo el valor obtenido de la base de datos.

Por este motivo, se introduce dentro de un bucle de tipo "*While*" la totalidad del cuerpo del código, incluyendo el protocolo *Try/Except/Pass*, con el objetivo de que se realicen los intentos de adquisición repetidamente (desatendiendo a los potenciales errores) hasta lograr la información deseada.

El código de adquisición se subdivide en dos bloques principales: el primer bloque son las acciones tomadas para la obtención correcta de los datos.

```
datagen = requests.get(URL1).json()

        for i in range(len(datagen)):
            if ID == datagen[i]["id"]:
                GEN = datagen[i]
                datagen = 0 #liberamos memoria
                print(GEN)
                return GEN
```

*Código 28: get_general_data(), primer bloque de adquisición, getGEN.py*

Para acceder a la información contenida dentro de la base de datos de Know Ice S.L., se emite al servidor una petición de tipo "*get*", dirigiéndola a la siguiente dirección URL (*URL1*): "*http://192.168.1.11:3000/api/knowice/ID/new*". Esta petición es redirigida por la API al router *GenRout.js* (Ver punto 3.1.2). Para ello, se hace uso del comando "*get*" perteneciente a la librería "*urequests*".

El servidor cierra dicha petición devolviendo la información general de todos los dispositivos dados de alta. Esta información se emite en formato JSON, sin embargo, el microcontrolador de Raspberry procesa la información recibida como un mensaje de texto.

Para poder procesar correctamente la respuesta del servidor, se debe convertir el mensaje recibido al formato JSON. Esto se hace incluyendo la función "*.json*()" al final del mensaje recibido "datagen". En este código, se incluye dicho comando al final de la propia solicitud "*get*".

De esta manera, la Raspberry es capaz de interpretar el mensaje, pudiendo navegar por sus apartados.

El mensaje de respuesta recibido por el servidor consiste en un vector el cual contiene en cada posición un cuerpo de la clase *objeto*. Los *objetos* son entidades propias del lenguaje de *JavaScript* que, desde un punto de vista práctico, se comportan como vectores cuyas posiciones en vez de estar enumeradas, poseen un título que se emplea para describir las propiedades de la información que contiene.

En este caso, la función "*get_general_data*()" realiza una criba del contenido en busca de un dispositivo de venta que haya sido dado de alta en la base de datos y cuyo contenido del atributo "*id*" sea idéntico al identificador único embebido en la memoria *Flash* del microcontrolador.

Para llevar a cabo dicho análisis, se hace uso de un bucle de tipo "*for*". Este bucle permite acceder individualmente a las entidades de tipo objeto del vector. El bucle se realiza a lo largo de la extensión completa del vector principal, a menos que se localice el dispositivo deseado.

Para ello se hace uso del comando "*len*()", el cual, al ser provisto de un vector, devuelve el número de posiciones que este contiene. Esto se combina con el comando "*range*()", que genera un vector de la longitud que se le haya indicado, incluyendo en cada posición el índice de esta.

En el interior del bucle *for* se ejecuta una comparación entre el atributo "*id*" del *objeto* que se compara y la variable "*ID*" definida al inicio de la función.

En el caso de que esta comparación de un resultado positivo, el *objeto* contenido en la posición del vector correspondiente se almacena en la variable "*GEN*", la cual a su vez es devuelta a la parte del código principal que ha hecho la llamada a la función "*get_general_data*()" en primer lugar.

Adicionalmente, se borra el contenido de la variable "*datagen*" que posee el contenido de la respuesta proporcionada por el servidor de Know Ice S.L.

El segundo bloque del programa se ejecuta exclusivamente cuando el microcontrolador no es capaz de encontrar un dispositivo de alta con la misma id.

Si tras la revisión del vector completo de "*datagen*" la condición de igualdad no ha resultado en un valor positivo a lo largo del recorrido, el programa no accede a la línea de código de *return*, haciendo que la función "*get_general_data*()" continúe ejecutándose.

Una vez fuera del bucle *for*, se define la variable "*post_data*". En ella se almacena un objeto en formato JSON con la siguiente estructura {'id': (…)}, completando el atributo *id* del objeto con el identificador de la propia Raspberry.

Este paquete de datos se envía mediante el protocolo HTTP de tipo *put* a la dirección definida anteriormente como *URL2*, correspondiendo a la siguiente dirección del servidor simulado de Know Ice S.L. http://192.168.1.11:3000/api/knowice/ID/new. Esta petición es redirigida por la API al *router* NewRout.js (ver punto 3.1.3), donde es procesada.

La nomenclatura de la petición de tipo *put* en micro Python posee la siguiente estructura: "requests.put(URL2, headers = {'content-type': 'application/json'}, data = post_data)".

La petición se llama con la subcategoría *put* de la función *requests*. El contenido de la función debe contener los siguientes apartados en orden: dirección URL donde redirigir la petición. Cabecero (*header*) de la petición, este debe definir el tipo de contenido que se emite, en este caso un mensaje de formato JSON. Y por último el contenido o cuerpo de la petición, almacenado en el atributo *data*.

El objetivo de esta función es el de poner a la disposición de un operario el id único de la Raspberry Pi Pico controladora del dispositivo de venta. De esta manera, un usuario hipotético puede acceder a la misma dirección descrita con anterioridad (http://...../knowice/ID/new) y emplear el contenido del objeto almacenado en esta dirección para completar los campos necesarios para realizar el alta del nuevo dispositivo, tal y como se muestra en el punto 3.1.2.4.

Una vez esta petición ha sido cerrada con éxito, remitiendo el servidor una respuesta exitosa con el código de estado asociado 200, el programa procede a la revisión periódica de la base de datos.

Para ello, se integra un nuevo bucle de tipo *while* anidado, en este se realiza la misma función que en el primer bloque, a través de peticiones de tipo *get* a la *URL1* del servidor en intervalos de 1 segundo.

Cada ciclo del bucle *while,* analiza el contenido de la base de datos en búsqueda de un dispositivo con la misma *id* mediante el mismo proceso estudiado en el primer bloque del programa, realizando una condición de igualdad con el atributo "*id*" de los objetos presentes en el vector *datagen*.

Una vez haya sido dado de alta el dispositivo, la Raspberry confirma la existencia de un dispositivo con un *id* idéntico al suyo. En este caso, el microcontrolador procede a almacenar el contenido del vector referente al dispositivo en la variable "*GEN*". Adicionalmente, se vuelve a borrar el contenido de la variable "*datagen*" para que no ocupe memoria valiosa en el dispositivo de control.

Por último, antes de terminar la ejecución del programa, la función realiza una nueva emisión de datos mediante protocolo *put* para normalizar el contenido del router GenRout.js. Para ello procede a repetir la petición enviada previamente a la dirección *URL2*, esta vez, sin embargo, conteniendo el siguiente mensaje en el cuerpo: "{'id': 'no se encuentran nuevos dispositivos'}".

### 3.2.5: ascii_hex.py

Como ya se explicó en el punto 3.2.4, el proceso de alta de los equipos emplea el router *NewRout.py* del servidor como intermediario.

Este proceso requiere la emisión de mensajes en formato JSON a una dirección concreta, estos mensajes tienen cuerpos con las siguientes características como ejemplo: "b'\xe6aL1\x1baL%'" "b'\xe6a\x18\xc4\xe3R@)'".

Sin embargo, tal y como se puede apreciar, los identificadores cuentan con una serie de caracteres poco convencionales.

Estos caracteres, aunque no suponen ningún problema a nivel de lenguaje a la hora de realizar operaciones en el propio microcontrolador (como, por ejemplo, las comparaciones de igualdad con otros identificadores que se realizarán durante el proceso de identificación), sí suponen un problema a la hora de su transmisión en protocolos web y HTTP.

Desgraciadamente, el protocolo de comunicación empleado en el entorno web, limita el tipo de contenido que puede transmitirse. En este caso, si se tratase de transmitir el primer *id* mostrado (b'\xe6aL1\x1baL%'), la Raspberry se encarga de transformarlo a formato JSON sin pérdidas. Sin embargo, durante el protocolo de emisión mediante protocolo HTTP, la información se ve distorsionada en el proceso de compresión y descompresión, dando como resultado el siguiente grupo de caracteres en el servidor: "  aL1\u001baL%".

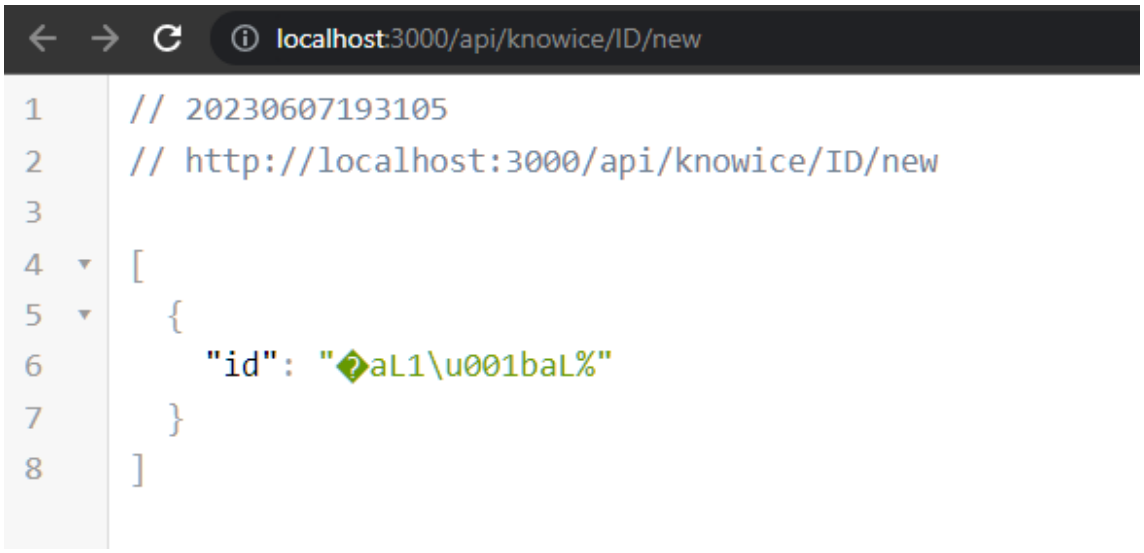



*figura 31 Resultado de la transmisión directa del identificador único de la Raspberry.*

El empleo de este mensaje "corrupto" para realizar el alta de un equipo de venta daría como resultado un dispositivo en la base de datos cuya id no corresponde con el de ningún microcontrolador, al mantener este su forma alterada durante la posterior adquisición de datos del microcontrolador.

```
>>> b'\xe6aL1\x1baL%' == 'aL1\u001baL%'
False
```

*figura 32: comparación de IDs fallida*

Por este motivo, se tuvo que buscar una alternativa al empleo del *id* embebido de la Raspberry en favor de una opción que permitiera su transmisión a través del servidor de Know Ice S.L.

Debido a la necesidad de generar un código fácil de implementar para los distintos dispositivos que se ponen a disposición del público, resulta poco práctico implementar un identificador a mano, exclusivo para cada dispositivo, previo a la puesta en marcha de la propia máquina de venta.

Por este motivo, se decidió continuar con el identificador único embebido en la memoria flash de cada Raspberry. Para solventar el programa, sin embargo, se decidió alterar el identificador mediante un protocolo lineal, integrado en el programa del microcontrolador.

Este programa debe alterar el identificador, de tal forma que este pueda ser transmitido mediante protocolo web sin ningún tipo de pérdidas.

Para conseguir esto, se estableció la conversión de los caracteres que conforman el identificador, a su equivalente hexadecimal en el código ASCII.

Dada la falta de un protocolo integrado con tal fin entre las librerías de micro Python, tras investigar exhaustivamente, finalmente se localizó un código similar a aquello que se necesitaba para hacer funcionar el código de forma adecuada.

El siguiente código pertenece a la autoría de Dave Hylands (nombre de usuario en github: *davehylands*), el cual puso a disposición una solución al problema de conversión de caracteres ASCII a su homólogo hexadecimal, compatible con la computación de micro Python.

El código que se muestra a continuación representa la única porción del proyecto que no ha sido escrito por el autor, habiendo sido tan sólo ligeramente modificado (modificaciones señalizadas en rojo para control de cambios) para cumplir con los propósitos del proyecto.

Esta es una de las pocas aportaciones externas al proyecto [HYLA20].

```
"""Provides the dump_mem function, which dumps memory in hex/ASCII."""

import sys
if sys.implementation.name == 'micropython':   # pragma: no cover
    import binascii

    def hexlify(buf):
        """Converts a binary string into its hex string representation
           with a space between each byte.
        """
        return binascii.hexlify(buf, ' ')
else:
    def hexlify(buf):
        """CPython's hexlify doesn't have the notion of a seperator character
           so we just do this the old fashioned way.
        """
        return bytes(' '.join(['{:02x}'.format(b) for b in buf]), 'ascii')

# pylint: disable=too-many-arguments
# pylint: disable=too-many-locals
# pylint: disable=too-many-branches

#para dejarlo bien, puse show_ascii y show_addr en false y añadí/quité un par de líneas
def dump_mem(buf, prefix='', addr=0, line_width=16, show_ascii=False,
             show_addr=False, log=print):
    """Dumps out a hex/ASCII representation of the given buffer."""
    if line_width < 0:
        line_width = 16
    if len(prefix) > 0:
        prefix += ':'
    if buf is None or len(buf) == 0:
        log(prefix + 'No data')
        return
    buf_len = len(buf)
    # Use a memoryview to prevent unnecessary allocations
    buf_mv = memoryview(buf)
```

```
line_ascii = ''
    ascii_offset = 0

    prefix_bytes = bytes(prefix, 'utf-8')
    prefix_len = len(prefix_bytes)
    if prefix_len > 0:
        prefix_len += 1    # For space between prefix and addr
    max_len = prefix_len
    if show_addr:
        max_len += 6
    hex_offset = max_len
    max_len += line_width * 3 - 1
    if show_ascii:
        ascii_offset = max_len + 1
        max_len += line_width + 1
    out_line = memoryview(bytearray(max_len))
    if prefix_len > 0:
        out_line[0:prefix_len-1] = prefix_bytes
        out_line[prefix_len-1:prefix_len] = b' '

    line_hex = out_line[hex_offset:hex_offset + (line_width * 3)]
    if show_ascii:
        # space between hex and ascii
        out_line[ascii_offset-1:ascii_offset] = b' '
        line_ascii = out_line[ascii_offset:ascii_offset + line_width]

    for offset in range(0, buf_len, line_width):
        if show_addr:
            out_line[prefix_len:prefix_len + 6] \
                = bytes('{:04x}: '.format(addr), 'ascii')
        line_bytes = min(buf_len - offset, line_width)
        line_hex[0:(line_bytes * 3)-1] \
            = hexlify(buf_mv[offset:offset+line_bytes])
        out_len = hex_offset + line_bytes * 3 - 1
        if show_ascii:
            if line_bytes < line_width:
                for i in range(line_bytes * 3 - 1, line_width * 3):
                    line_hex[i:i+1] = b' '
            #line_ascii[0:line_bytes] = buf_mv[offset:offset + line_bytes]

            for i in range(line_bytes):
                char = line_ascii[i]
                if char < 0x20 or char > 0x7e:
                    line_ascii[i] = ord('.')
            out_len = ascii_offset # + line_bytes                #linea modificada para que no ponga el original
        #log(bytes(out_line[0:out_len]).decode('utf-8'))         #línea eliminada para que no haga print
        addr += line_width
        return(bytes(out_line[  :out_len]).decode( utf-8'))      #línea añadida

"""
https://github.com/dhylands/bioloid3/blob/master/bioloid/dump_mem.py
dhylands/bioloid3
"""
```

*Código 29: código dump_mem.py de davehylands, ascii_hex.py*

Debido a su complejidad, no se van a explicar los pormenores de las funciones integradas en el propio código. Tanto las líneas eliminadas como los atributos modificados en el cabecero de la función fueron realizados con el objetivo de cambiar la respuesta recibida por esta de la siguiente manera:


*figura 33: respuesta de la función dump_mem() previa a las modificaciones, ascii_hex.py*

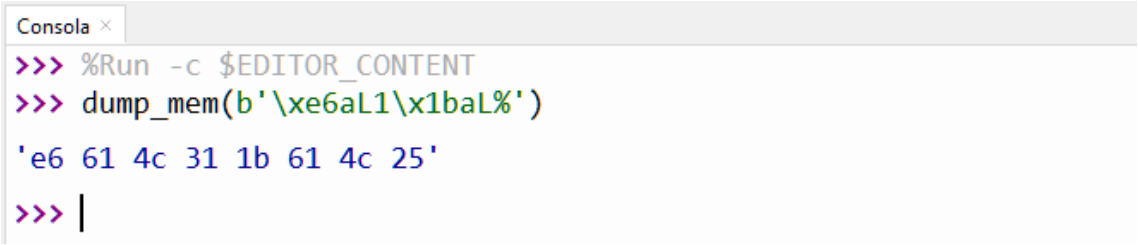



*figura 34: respuesta de la función dump_mem() posterior a las modificaciones, ascii_hex.py*

La última línea de código se añade con el objetivo de devolver el resultado en formato de texto al código origen de la llamada de la función.

Si estudiamos la respuesta recibida por la función, podemos observar que el número de códigos ASCII devueltos no es coherente con la cantidad de caracteres del identificador original.

Esto se debe a que la propia función "*dump_mem*()" realiza un proceso similar a la conversión efectuada en durante los protocolos web. Esto se debe a la confusión de algunos caracteres como elementos del lenguaje HTML.

Si se realiza una conversión inversa, haciendo uso de un conversor web hexadecimal ASCII a texto [BINA23], se puede apreciar cómo la función "*dump_mem*()" interpreta algunos de los elementos del mensaje.

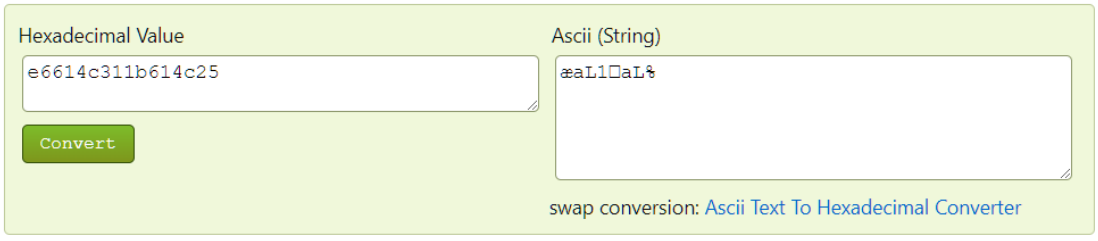



*figura 35: reversión del proceso realizado por la función dump_mem()*

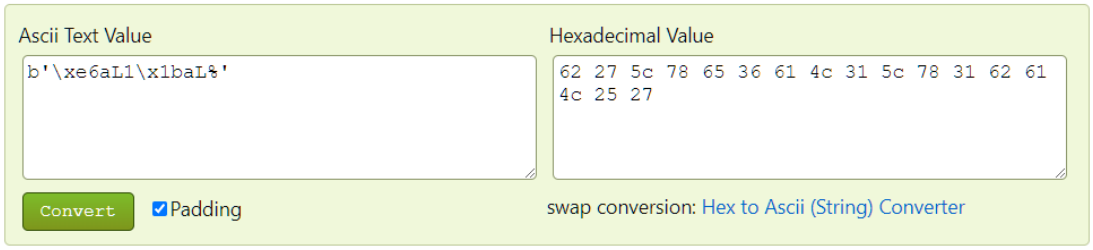



*figura 36: conversión a elemento ASCII en hexadecimal paso a paso.*

Como se puede observar en la figura 35, si se revierte la conversión hecha por la función "*dump_mem*()", el resultado obtenido resulta no ser equivalente al texto de origen, dando como resultado la siguiente respuesta "æaL1aL%".

Sin embargo, si se realiza la conversión de texto a hexadecimal ASCII, transformando cada elemento del identificador por separado, el resultado es equivalente al mostrado en la figura 36.

A pesar de esto, aun no tratándose de una conversión ideal del identificador extraído de la memoria flash del microcontrolador, el resultado obtenido consiste en una serie de caracteres capaces de ser transmitidos mediante protocolo web sin dar lugar a pérdidas.

Al mismo tiempo, el resultado de las operaciones realizadas por la función "*dump_mem*()" consiste en un identificador único e irrepetible de placa a placa, cumpliendo con el objetivo por el cual se decidió emplear dicho identificador en primer lugar.

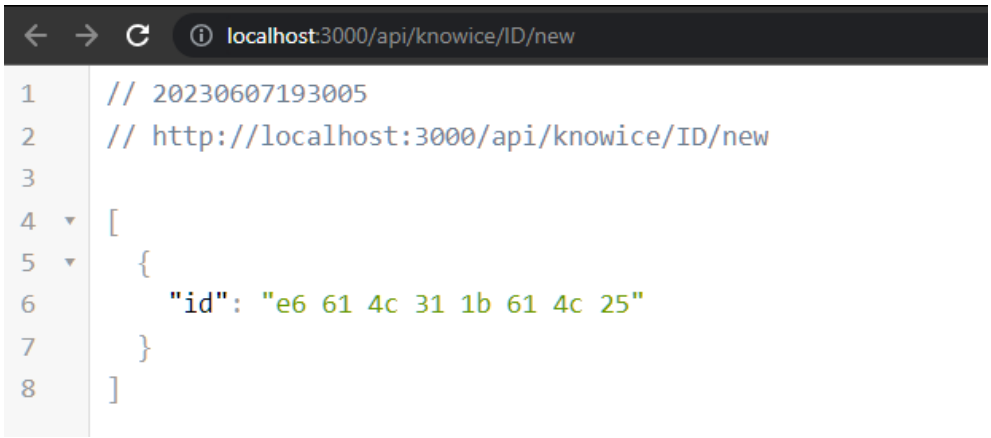



*figura 37: resultado de la emisión exitosa de id al router NewRout.js*

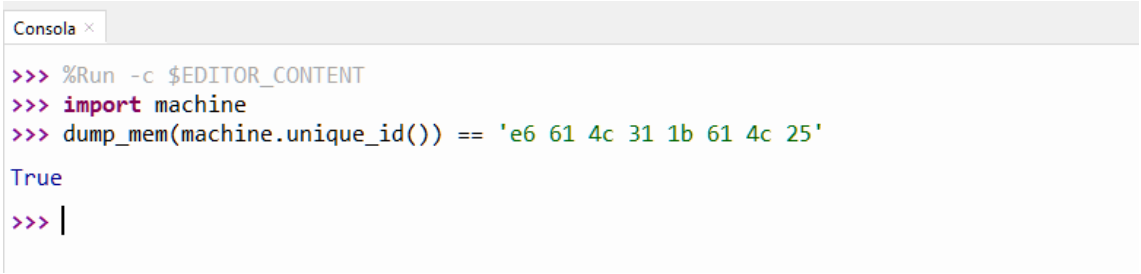



*figura 38: comparación de IDs exitosa*

### 3.2.6: varios.py

El último programa del que consta este proyecto es el de "*varios.py*".

Este documento está conformado por las funciones de empleos diversos que permiten el funcionamiento correcto del dispositivo. Entre estas funciones se encuentran las comunicaciones con la plataforma de pago y otros empleos diversos que hacen reconocibles los mensajes recibidos por esta.

#### *3.2.6.1: definiciones del programa varios.py*

El programa "*varios.py*" se emplea para cumplimentar una variedad de servicios tanto en el resto de los programas como en las funciones integradas dentro de "*varios.py*".

A continuación, se muestran las definiciones realizadas en torno a las funciones de las cuales se habla más adelante.

```
import urequests as requests
import ujson
import time
import LED_wifi
```

*figura 39: importación de librerías y archivos, varios.py*

Las librerías importadas en el archivo "*varios.py*" son similares a las que ya se han analizado hasta ahora, estando compuestas por las librerías *urequests* y *ujson* necesarias para la comunicación mediante protocolo HTTP. También se incluye la librería *time*, para la realización de pausas en el programa.

Por último, se incorpora el archivo del programa "*LED_wifi.py*", el cual será empleado para la verificación de conexión WLAN más adelante.

```
"""
  OBJETOS
"""
header= {'content-type': 'application/json'}
```

*figura 40: objeto header, varios.py*

```
autoriz={
  "pinpad": "*",
  "opType": "auth",
  "createReceipt": False,
  "executeOptions": {
    "method": "polling"
    },
  "language": "es",
  "requestedAmount": 420,
  "requireConfirmation": False,
  "transactionReference": "nose"
  }
```

*figura 41: objeto autoriz, varios.py*

```
autclose={
  "pinpad": "*",
  "opType": "authClose",
  "cardNumberHashDomain":"branch",
  "createReceipt": False,
  "executeOptions": {
    "method": "polling"
    },
  "language": "es",
  "requestedAmount": 220,
  "requireConfirmation": False,
  "transactionReference": "nose",
  "operationDetails":{
    "previousOperationNumber": 123456
    }
  }
```

*figura 42: objeto autclose, varios.py*

```
poll={
  "pinpad": "*"
  }
```

*figura 43: objeto poll, varios.py*

En las figuras mostradas se representan los paquetes de datos que se emplean para mediar las comunicaciones con la plataforma de pagos de Paytef.

Estos cuerpos definidos al final del archivo pueden ser empleados tanto en las funciones que se integran en el propio código de "*varios.py*" como en el programa principal de "*main.py*" con el fin de simplificar los procesos de autorización y venta.

El primer cuerpo definido, "*header*" (figura 40), define el cabecero del cuerpo de las peticiones, el cual se repite para todos los procesos de comunicación con la plataforma de pago. De esta forma se simplifica la nomenclatura de las funciones mostradas en el código más adelante.

El siguiente cuerpo que se define es el de "*autoriz*" (figura 41), este es un objeto que se adjunta a la petición de tipo *post* para inicializar el proceso de autorización bancaria.

Asimismo, se encuentra el cuerpo "*requestedAmount*". Este representa el saldo mínimo necesario para autorizar la apertura de compuertas del equipo de venta de hielo. Esta cantidad es "congelada" por parte del banco, a petición de la plataforma de pago, hasta que se libere mediante el cierre de la autorización.

El siguiente objeto, "*authclose*" (figura 42), es el adjuntado en el cuerpo de la petición de tipo *post* que completa la venta, cerrando el protocolo de venta y liberando el fondo congelado por la petición de autorización previa.

Otro de los elementos que componen este objeto es "*requestedAmount*", en el caso de esta petición, este elemento informa a la plataforma del valor a restar de la cuenta bancaria, equivalente al valor de los bienes sustraídos.

Tanto el atributo de "*requestedAmount*" empleado en la petición de autorización como el del cierre de autorización son valores expresado en céntimos de euro.

Además, se define el cuerpo de "*poll*" (figura 43), este es emitido mediante petición de tipo *post* para obtener una respuesta que defina el estado actual del dispositivo de pago y el de sus procesos.

```
"""
   URL
"""

URL_start = "http://192.168.1.16:8887/transaction/start"

URL_poll = "http://192.168.1.16:8887/transaction/poll"
```

*figura 44: definición de URLs, varios.py*

Por último, en la figura 44 se definen las direcciones URL a las que se emiten las peticiones de tipo *post*. Tanto las peticiones de autorización como de cierre son emitidas a la dirección de la "*URL_start*", mientras que las peticiones de tipo *poll* se remiten a la direcciòin "*URL_poll*".

### *3.2.6.2: auth_ini()*

La función *auth_ini*() inicializa el proceso de autorización bancaria, realizando la comprobación de saldo en la cuenta asociado a la tarjeta de crédito del usuario.

```
def auth_ini(reten):#esta función no trabaja bajo la premisa de un reinicio, por lo que de momento no está a prueba
                    de apagones
  try:        #para estarlo lo primero que debería hacer sería un poll y leer si una tarjeta fue aprobada, también tener
               en cuenta el estado de la puerta y la diferencia en las galgas etc etc
    global autoriz, header, poll
    autoriz["requestedAmount"] = reten*100
    response = requests.post(URL_start, headers = header, data = ujson.dumps(autoriz))
    time.sleep(10/1000)
    if response.status_code == 200:
      response = requests.post(URL_poll, headers = header, data = ujson.dumps(poll))
      time.sleep(10/1000)
      resp = limpia_JSON(response.text)
      while(resp["info"]["transactionStatus"]  ==  "waitingForUser"  or  resp["info"]["transactionStatus"]  ==
"processing"):
        time.sleep(200/1000)
        response = requests.post(URL_poll, headers = header, data = ujson.dumps(poll))
        resp = limpia_JSON(response.text)
      if(resp["result"]["failed"] == True):
        time.sleep(200/1000)
        auth_ini(reten)
        #error (o no se leyó ninguna tarjeta)
      elif(resp["result"]["approved"] == False):
        time.sleep(1)
        auth_ini(reten)
        #saldo insuficiente
      else:
        time.sleep(20/1000)
        print("victoria")
        return resp["result"]["paytefOperationNumber"]
        #todo salió perfect, tenemos número de operación que usaremos para cerrar cobrando
```

```
else:
        time.sleep(20/1000)
        auth_ini(reten)
        #si el código de respuesta no es 200 que vuelva a empezar
  except:
    if(LED_wifi.LAN_status == False):
      return "Error"
    else:
      auth_ini(reten)
      #si es que no hay internet, que devuelva error y ya lo gestiono en el main
      #si no que vuelva a empezar
  else:
    time.sleep(0.5)
    #si todo salió oki que se tome medio segundo de descanso
```

*Código 30: función auth_ini(), varios.py*

La función "*auth_ini*()" se basa principalmente en protocolos de comunicación HTTP, por lo que para prevenir potenciales errores catastróficos en la ejecución el programa, la función se ejecuta dentro de las funciones de los comandos *try/except/else.*

La primera acción tomada dentro del cuerpo de la función consiste en la importación de las variables globales que se emplean para la ejecución del programa, estas son "*autoriz*", "*header*" y "*poll*".

A continuación, el programa altera el apartado "*requestedAmount*" del objeto de autorización "*autoriz*", integrando en este el contenido del atributo de llamada a la función, "*retén*". Se da por hecho que este atributo se ha aportado en unidades de euro, por lo que se multiplica su valor por 100 para corresponder al formato de la plataforma de pago, la cual trabaja en céntimos.

El nuevo objeto "*autoriz*" se incorpora al cuerpo de la petición de tipo *post*, transformándose al formato JSON mediante el empleo de la función "*ujson.dumps*(…)". La dirección de envío corresponde a la de "URL_start", y se completa el apartado de *header* con el objeto definido con su mismo nombre.

La respuesta del servidor a la petición se almacena en la variable "*response*". Esta variable contiene el objeto de respuesta codificado en bloque, mostrando el mensaje <*Response Object at …*> cuando se solicita que se muestre (figura 45).

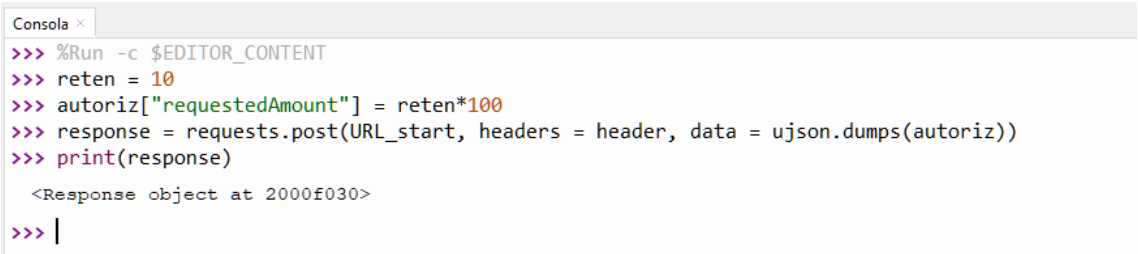



*figura 45: objeto response, auth_ini(), varios.py*

Por lo tanto, para navegar entre sus apartados se usan los comandos "*.status_code*" y "*.text*", dando como respuesta los siguientes resultados

```
>>> response.status_code
200
>>> response.text
'c7\r\n{\n  "info" : {\n    "message" : "Transacci\xf3n en proceso",\n    "opType" : "auth",\n
rsion" : "2022.12.280258"\n}\r\n0\r\n\r\n'
>>>
```

*figura 46: comandos .status_code y .text, auth_ini(), varios.py*

Como se puede apreciar, el contenido devuelto por "*response.text*" no tiene un formato que permita procesarlo directamente, por lo que es necesario realizar un proceso de "limpieza" para poder interpretar las respuestas del servidor. Este proceso se estudia con mayor profundidad en el punto 3.2.6.3 (limpia_JSON()).

Una vez obtenida la respuesta del servidor, se desdoblan las acciones a tomar en función del código HTTP adjunto a la respuesta del servidor.

En el caso de suceder un error en el servidor durante el procesamiento de la petición realizada por la plataforma de pago, se realiza una pausa de 20 milisegundos, tras los cuales, la función "*auth_ini*()" realiza una llamada anidada a sí misma para volver a iniciar el proceso de autorización desde el principio.

Esto se hace llamando a la función "*auth_ini*()", incorporando el valor de "*reten*" a los atributos de llamada.

En el caso contrario, si el código de respuesta es 200, significa que el servidor de Paytef ha recibido la orden de autorización y la ha remitido a la plataforma de pago asociada, en dicho caso el programa continuará ejecutándose.

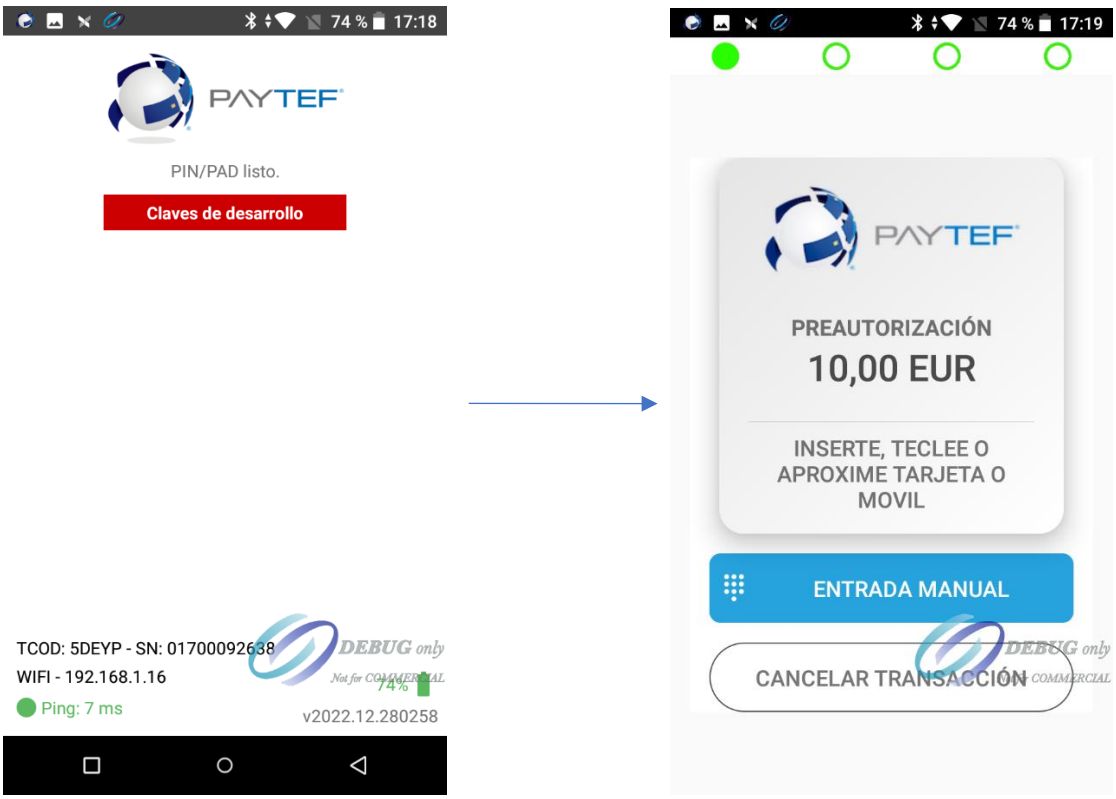



*figura 47: pantalla del dispositivo A50, de PAX, durante el proceso de autorización*

La primera acción una vez se ha iniciado el proceso de autorización, consiste en la emisión de una petición *poll*. Esta es emitida mediante protocolo *post* y, una vez recibida por el servidor, este remite una respuesta tipo objeto, descriptiva de su estado, y el estado de la última petición recibida por el servidor.

Para realizar una petición de tipo *poll* al servidor, se hace una petición *post*, adjuntando como cuerpo el objeto definido como "*poll*" (figura 43) y se envía a la dirección descrita en la variable "*URL_poll*" (figura 44).

La respuesta de esta petición se almacena en la variable "*response*", cuyo contenido se procesa gracias a la función "*limpia_JSON*()" (ver punto 3.2.6.3).

Una vez hecho esto, el programa entra en un bucle, solicitando información sobre el estado de la operación de autorización al servidor. Cada 200 microsegundos, el microcontrolador repite la petición y procesa el contenido de su respuesta a la espera de actualizaciones.

Durante el proceso de autorización, mientras la plataforma de pago está a la espera de que se le introduzca una tarjeta de crédito para su lectura, las respuestas recibidas tienen una estructura similar a la siguiente:


```
{
 "info" : {
  "sessionID" : "{04EEDDC7-FDB5-4F07-8B65-650D659BD13C}",
  "cardStatus" : "finished",
  "opType" : "auth",
  "requestedAmount" : 600,
  "tcod" : "5DEYP",
  "transactionStatus" : "waitingForUser",
  "transactionReference" : "nose"
 },
 "version" : "2022.12.280258"
}
```


*figura 48: : respuesta a petición poll durante el proceso de autorización, previo al inicio de espera, auth_ini()*


```
{
 "info" : {
  "sessionID" : "{91F57518-AFB6-4CA1-9F74-08A3D5472560}",
  "cardStatus" : "waitingForCard",
  "opType" : "auth",
  "requestedAmount" : 6969,
  "tcod" : "5DEYP",
  "transactionStatus" : "waitingForUser",
  "transactionReference" : "nose"
 },
 "version" : "2022.12.280258"
}
```


*figura 49: respuesta a petición poll durante el proceso de autorización, previo a la lectura de una tarjeta, auth_ini()*

Durante la espera, el microcontrolador recibe dos posibles respuestas a la petición *poll.* Dentro de las cuales, el objeto "*info*" anidado contiene información sobre el estado de la petición. La primera de estas posibles respuestas (figura 48) contiene el atributo "*cardStatus*" con el valor asociado "*finished*", y el atributo "*transactionStatus*" con el valor "*waitingForUser*".

Estas respuestas se dan durante las primeras peticiones poll al servidor tras enviar la orden de autorización. Una vez procesada la orden, el servidor responde con un objeto similar al mostrado en la figura 49, modificando el atributo "*cardStatus*" de tal forma que muestre el valor "*waitingForCard*".

Si durante el proceso de autorización, se introdujera una tarjeta de crédito, el objeto devuelto por la petición *poll* sería similar al siguiente*:*


```
{
 "info" : {
  "sessionID" : "{9077160C-7E60-45C6-9EB5-8B175AAC1CD2}",
  "cardStatus" : "processing",
  "opType" : "auth",
  "requestedAmount" : 6969,
  "tcod" : "5DEYP",
  "transactionStatus" : "processing",
  "transactionReference" : "nose"
 },
 "version" : "2022.12.280258"
}
```


*figura 50: respuesta a petición poll durante el proceso de autorización, lectura de una tarjeta, auth_ini()*

Dado que estas respuestas suponen un estado de espera o procesamiento, el programa permanece a la espera, emitiendo peticiones *poll* al servidor cada 200 microsegundos mientras el atributo "*transactionStatus*" contenido dentro del objeto anidado "*info*" posea el valor de "*waitingForUser*" o "*processing*". Esto se debe a que una respuesta con cualidades distintas a las descritas significa que el estado de espera de la plataforma de pago ha finalizado y requiere que se tomen acciones. Las posibles respuestas una vez salido del bucle de espera son las siguientes (las respuestas han sido reducidas debido a su gran volumen):

```
{
 "info" : {
  "sessionID" : "{91F57518-AFB6-4CA1-9F74-08A3D5472560}",
  "cardStatus" : "finished",
  "opType" : "auth",
  "requestedAmount" : 6969,
  "tcod" : "5DEYP",
  "transactionStatus" : "finished",
  "transactionReference" : "nose"
 },
 "result" : {
  "approved" : false,
  "failed" : true,
  "resultText" : "Operacion Cancelada (Code = 2)", (.....)
```


*figura 51: Error de autorización, tiempo excedido*


```
{
 "info" : {
  "sessionID" : "{9077160C-7E60-45C6-9EB5-8B175AAC1CD2}",
  "cardStatus" : "finished",
  "opType" : "auth",
  "requestedAmount" : 6969,
  "tcod" : "5DEYP",
  "transactionStatus" : "finished",
  "transactionReference" : "nose"
 },
 "result" : {
  "approved" : false,
  "failed" : true,
  "resultText" : "IMPORTE VENTA NO AUTORIZADO", (.....)
```


*figura 52: error de autorización, saldo insuficiente*


```
{
 "info" : {
  "sessionID" : "{B0CEBEF0-B386-4F39-A782-2C138A5AC517}",
  "cardStatus" : "finished",
  "opType" : "auth",
  "requestedAmount" : 777,
  "tcod" : "5DEYP",
  "transactionStatus" : "finished",
  "transactionReference" : "nose"
 },
 "result" : {
  "approved" : false,
  "failed" : false,
  "resultText" : "190 - Simulacion denegada (Entero Impar)", (.....)
```


*figura 53: error de autorización, error genérico*

```
{
 "info" : {
  "sessionID" : "{0888F9EA-758D-4035-899F-E2A7304994E0}",
  "cardStatus" : "finished",
  "opType" : "auth",
  "requestedAmount" : 420,
  "tcod" : "5DEYP",
  "transactionStatus" : "finished",
  "transactionReference" : "nose"
 },
 "result" : {
  "approved" : true,
  "failed" : false,
  "resultText" : "000 - Simulacion aprobada", (.....)
```


*figura 54: autorización aceptada*

Para obtener estos resultados, se ha hecho uso de las características del entorno de simulación dispuesto por Paytef. Esta API está construida de tal forma que, en función de la cantidad de dinero especificada en la solicitud de autorización, ante la lectura de una tarjeta de crédito, el servidor envía una posible respuesta.

Las condiciones descritas son las siguientes:

- Para obtener como resultado una operación aprobada ("000 - Simulación Aprobada"), se deben solicitar operaciones con importes pares en su parte entera (en este caso 4.20€).
- Para obtener como resultado una operación denegada ("190 – Simulación denegada"), se deben solicitar operaciones con importes impares en su parte entera (en este caso 7.77€).
- Por último, para obtener como resultado una operación denegada por saldo insuficiente ("IMPORTE DE VENTA NO AUTORIZADO") , basta con solicitar un importe superior al que pueda suministrar la tarjeta de crédito.

Para procesar las respuestas recibidas, se integran los siguientes condicionales consecutivos:

- Si el valor del atributo "*result.failed*" es *True*, significa que ha sucedido un error en el proceso de autorización, potencialmente provocado por exceder el tiempo destinado a la operación (en torno a un minuto). En este caso se procede a repetir la operación, llamando de nuevo a la función "*auth_ini*()" proporcionando el valor de "*reten*" a los atributos de llamada.
- En el caso contrario, si el atributo "*result.failed*" es *True*, se analiza el atributo "*result.approved*". Si este tiene valor *False*, significa que no ha sucedido ningún error, pero no pudo aprobarse la operación, potencialmente debido a saldo insuficiente. En este caso se procede a repetir una vez más el proceso de llamada de la función "*auth_ini*()".

- Por último, si el valor del atributo "*result.approved*" es *True*, significa que la operación ha sido aprobada. En cuyo caso la función llega a su fin, devolviendo el valor perteneciente al atributo "*result.paytefOperationNumber*" al programa "*main.py*".

De aprobarse la operación, tal y como se acaba de describir, se inicia la secuencia de venta, permitiendo que el usuario adquiriera los bienes deseados.

Una vez finalizado este proceso, el microcontrolador cierra la petición, aún abierta, de autorización. Para ello, el microcontrolador debe emitir mediante protocolo *post* al servidor, una petición de cierre de autorización. Esto se hace adjuntando al cuerpo del mensaje el objeto definido como "*autclose*" (figura 42).

Este objeto debe modificarse para que el atributo "*previousOperationNumber*" ubicado dentro del objeto "*operationDetails*" contenga el número de la operación el cual autorizó el proceso ("*result.paytefOperationNumber*"). Además de modificar el atributo "*requestedAmount*" con el coste de los bienes sustraídos (en céntimos).

#### *3.2.6.3: limpia_JSON()*

La respuesta contenida en "*response.*text" (figura 46, punto 3.2.6.2) tiene la siguiente estructura cuando se analiza:

```
{
  "info" : {
    "message" : "Transacción en proceso",
    "opType" : "auth",
    "started" : true,
    "sessionID" : "{9E6F7E30-FBFB-41EC-9076-1703940309F3}"
  },
  "version" : "2022.12.280258"
}
```

*figura 55: cuerpo de la respuesta del servidor*

Sin embargo, la Raspberry Pi Pico no cuenta con un motor que le dé formato al texto recibido vía protocolo web, por lo que no es capaz de procesar los comandos ASCII. Esto resulta en el microcontrolador interpretando la respuesta del servidor de la siguiente forma:

```
'c7\r\n{\n  "info" : {\n    "message" : "Transacci\xf3n en proceso",\n    "opType" : "auth",\n
"started" : true,\n      "sessionID" : "{9E6F7E30-FBFB-41EC-9076-1703940309F3}"\n
},\n  "version" : "2022.12.280258"\n}\r\n0\r\n\r\n'
```

*figura 56: cuerpo de la respuesta interpretado por la Raspberry*

Estos comandos (\r \n \n0) corresponden a instrucciones dedicadas a representar el texto en un entorno web, sin embargo, con tal de poder procesar su contenido, el programa necesita eliminar estos caracteres. Con este fin se programó la función "*limpia_JSON*".

```
def limpia_JSON(resp):
    i = 1
    text = resp[0]

    while(resp[i]!="{"):
        text+=resp[i]
        i=i+1

    resp = resp.replace(text,"")
    i= len(resp)-1
    text = "}"

    while(resp[i]!="}"):
        i-=1
    i+=1

    while(i<=len(resp)-1):
        text+=resp[i]
        i+=1
    resp = resp.replace(text,"}")
    resp = resp.replace("\n","")
    resp = ujson.loads(resp)
    return resp
```

*Código 31: limpia_JSON, varios.py*

Cando esta función es llamada, lo hace recibiendo el contenido de la respuesta del servidor en su parámetro "*resp*".

Ya que el contenido real del cuerpo del mensaje consiste en un objeto, este está comprendido entre corchetes "{…}". Por lo tanto, la primera acción que se toma es eliminar aquellos caracteres que se encuentran fuera de los corchetes principales del mensaje.

Para ello, primero se realiza una lectura del mensaje desde el primer carácter en adelante. El lenguaje de micro Python permite hacer esto de manera sencilla, ya que interpreta las variables de texto como vectores, los cuales contienen un carácter por posición.

El primer bucle *while* acumula, carácter a carácter, el contenido del vector recibido "*resp*" en la nueva variable definida como "*text*". Realiza esta función hasta llegar a la posición del vector "*resp*" cuyo contenido es el símbolo de apertura de corchetes ({), significando que se ha llegado al inicio del cuerpo del objeto.

Ya con el vector "*text*" conteniendo el primer grupo de caracteres, se procede a eliminar estos del cuerpo del mensaje.

Para ello se utiliza el comando propio de micro Python, "*.replace*()". Este comando se emplea acoplándolo al final del vector que se desea modificar, y recibe dos valores, siendo el primero el carácter o grupo de caracteres objeto de la modificación, y el valor por el cual se pretenden sustituir.

Ya que en este caso se desea eliminar todo hilo de caracteres que correspondan a la variable "*text*", el cuerpo por el que se sustituyen es nulo ("").

A continuación, se procede a realizar la misma función en el otro extremo del mensaje. Primero se inicializa la variable de "*text*" como "}", y el contador empleado (*i*) se inicializa en el que sería la última posición de la variable "*text*" (len(text)-1).

A continuación, se procede a analizar el vector de forma regresiva mediante un bucle *while*, finalizando una vez que este haya encontrado el símbolo de cierre de corchetes, ubicado al final del objeto.

Con el contador "*i*" indicando la posición del primer elemento a eliminar, se inicia de nuevo mediante un bucle *while* un repaso progresivo del vector "*resp*", desde la posición indicada hasta el final del vector. Durante este proceso, se acumula el contenido del vector en la variable "*text*".

Una vez finalizado el bucle, se repite la eliminación de todo aquel grupo de caracteres que corresponda al acumulado en el vector "*text*", haciendo uso una vez más de la función "*.replace*()".

A continuación, para eliminar aquellos comandos ASCII repartidos por el resto del cuerpo del mensaje, se realiza una eliminación de los comandos de salto de línea (\n) sobrantes en el mensaje.

Por último, para poder navegar por el cuerpo del mensaje con facilidad, se convierte la variable "*resp*" (ya limpia de comandos ASCII) a formato JSON, haciendo uso de la función "*loads*" perteneciente a la librería "*ujson*".

Una vez hecho esto, se retorna la variable "*resp*", ahora libre de caracteres innecesarios y en formato JSON, al código que haya originado la llamada de la función en primer lugar.

```
>>> response = limpia_JSON(response.text)
>>> print(response)
  {'version': '2022.12.280258', 'info': {'started': True, 'sessionID': '{9E6F7E30-FBFB-41EC-9076
```

*figura 57: resultado de la función limpia_JSON(). varios.py*

## 3.3 Dispositivo de balanceo

Durante el proceso de instalación de los equipos, uno de los aspectos más importantes es el nivelado de la máquina frigorífica.

Debido a que el sistema elegido para la medida de peso consiste en sustituir los puntos de apoyo del equipo por células de carga, resulta imprescindible para la calidad de las medidas, asegurarse de que todas las células cargan con un peso similar en el momento de la instalación.

Por este motivo, la empresa promotora de este proyecto, Know Ice S.L., solicitó el desarrollo de un dispositivo capaz de facilitar el proceso de equilibrado durante la instalación.

Para ello, se inició el proceso de prototipado de un sistema capaz de medir las cuatro células de peso que conforman el sistema de medida de la cámara frigorífica, empleando para ello un circuito de electrónica analógica basada en amplificadores operacionales.

El dispositivo debe de ser capaz de medir las cuatro células por separado en un estado de descarga de peso, para posteriormente medir la diferencia de voltaje una vez apoyado el equipo sobre el suelo. Esto se hace mediante un programa ideado y desarrollado específicamente para este proyecto, a través del cual se representa el porcentaje de carga de cada una de las células.

Con esta información, el operario puede manipular las células con el criterio necesario, elevando de nuevo la máquina frigorífica y modificando la extensión de los apoyos. Dichos apoyos, consisten en una plataforma protegida por una goma, unida a la célula a través de una articulación con un vástago roscado.

Mediante el roscado/desenroscado de este vástago, el operario es capaz de desplazar la carga entre los distintos apoyos hasta conseguir el resultado deseado.

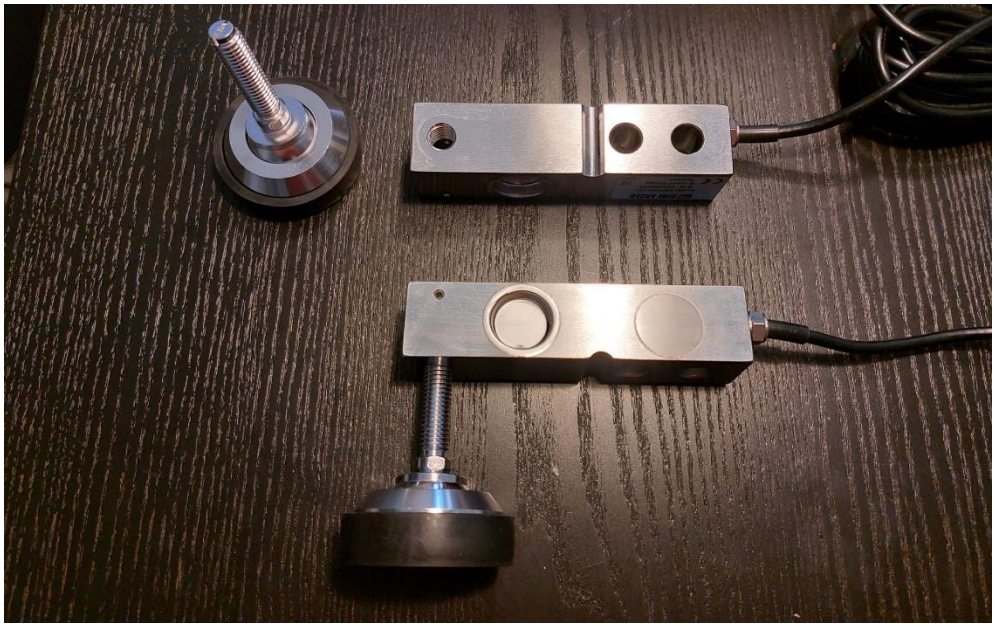


*figura 58: sistema de puntos de apoyo, células de carga*

Haciendo uso de la herramienta de AutoCAD, se inició el desarrollo de un circuito, diseñado con el objetivo de obtener un prototipo funcional sobre una placa PCB.

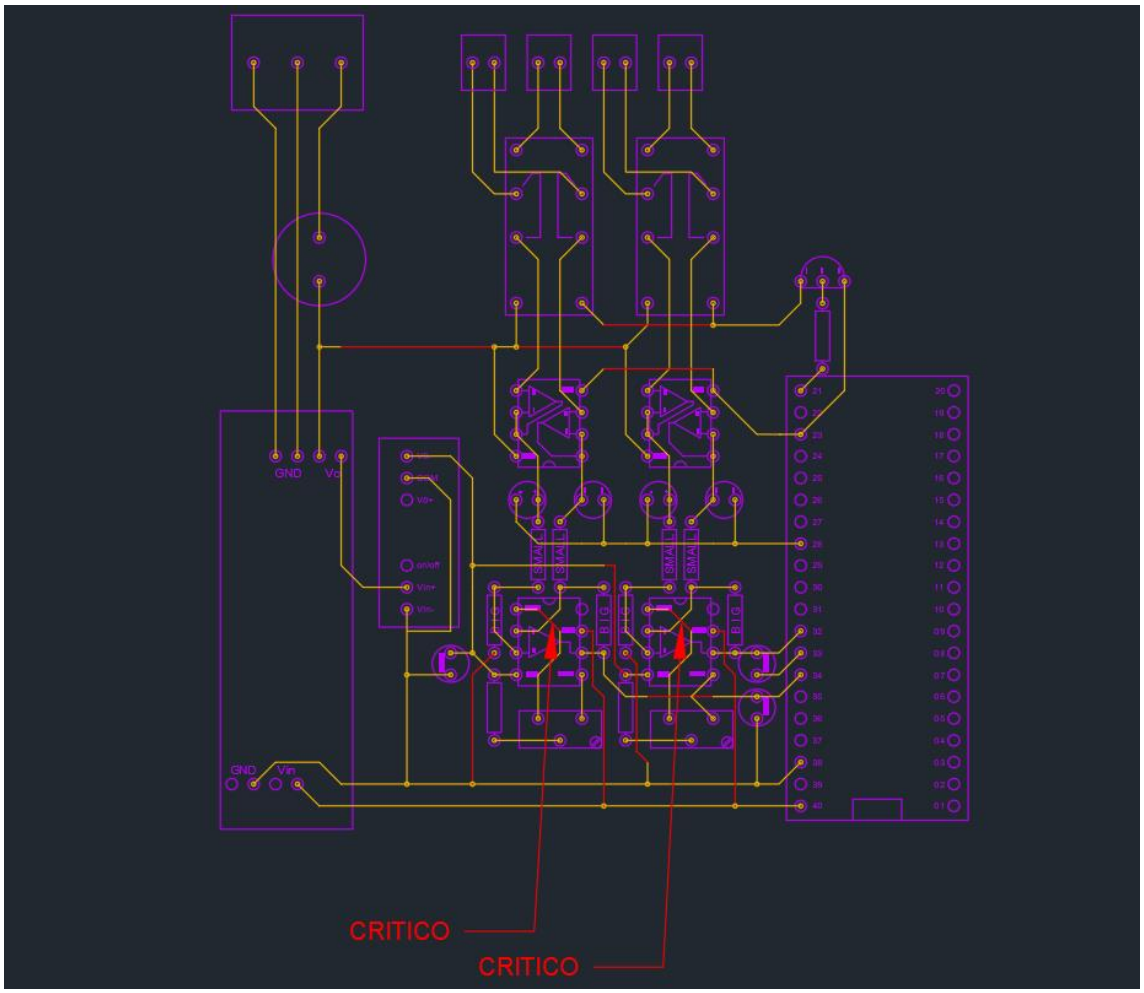



*figura 59: diseño inicial de la PCB, dispositivo de balanceo*

### 3.3.1: Circuito

En la figura 59 se muestra desde una perspectiva superior, la distribución de los distintos componentes que conforman el circuito de medida de las células de carga.

Tanto la placa PCB (100x100mm 35x37 perforaciones) como el resto de los componentes de esta, han sido diseñados a escala en autocad para obtener un diseño del producto final fiable.

Las conexiones esbozadas en el plano han sido codificadas por color, mostrándose en amarillo aquellos carriles que se integran en la parte inferior de la placa (en la cara contraria a los componentes) y en rojo aquellos que se deben completar por la parte superior, con tal de evitar posibles cruces entre ellos.

A continuación, se desarrolla el funcionamiento del diseño inicial del circuito de medida, así como sus distintos sistemas y componentes.

#### *3.3.1.1: sistema de control y alimentación*

El sistema de control y medida principal consiste en una placa programable Raspberry Pi Pico, similar a la empleada en el diseño de control de la cámara frigorífica. Sin embargo, el modelo empleado para el control del dispositivo de balanceo no cuenta con el módulo

de comunicación inalámbrica, ya que no es necesario para el funcionamiento de este dispositivo.

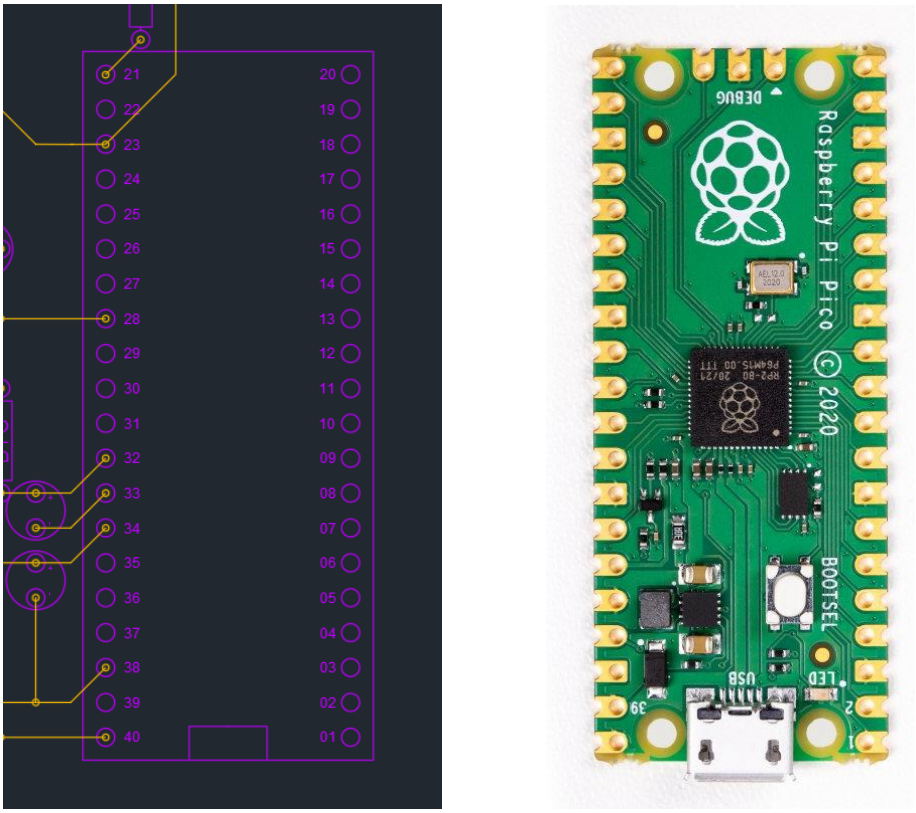



*figura 60:microcontrolador Raspberry Pi Pico, dispositivo de balanceo*

En el diseño inicial que se muestra, los puertos de la Raspberry empleados son los siguientes:

- 40: VBUS, esta es una salida directa del tren de alimentación de la propia Raspberry, conectando con el pin de alimentación del puerto micro USB. Este puerto se emplea para la alimentación de diversos elementos en el resto del dispositivo.
- 38/33/28/23: GND, tierra común empleada entre los distintos equipos.
- 34: GP28 & ADC2, este puerto se emplea para la digitalización de la señal analógica proporcionada por la unidad de medida A.
- 32: GP27 & ADC1, este puerto se emplea para la digitalización de la señal analógica proporcionada por la unidad de medida B.
- 21: GP16, este puerto lógico se emplea para la activación de los conmutadores de señal.
- 20: GP15, (NO MOSTRADO) este puerto lógico se emplea para la lectura del pulsador empleado en el control de la interfaz
- 01/02: GP0 I2C0 SDA / GP1 I2C0 SCL (NO MOSTRADO) estos puertos se utilizan para la comunicación entre la Raspberry y la pantalla LCD encargada de mostrar al usuario los resultados de la medida a través de la interfaz.

### 3.3.1.2: transformadores de tensión

El circuito de amplificación requiere diversos niveles de voltaje para realizar sus funciones.

El primer generador de voltaje consiste en un *boost* de tensión con una salida fijada a 12Vcc. Este se alimenta desde el tren de 5Vcc de la Raspberry, y se emplea para la alimentación de diversos equipos.

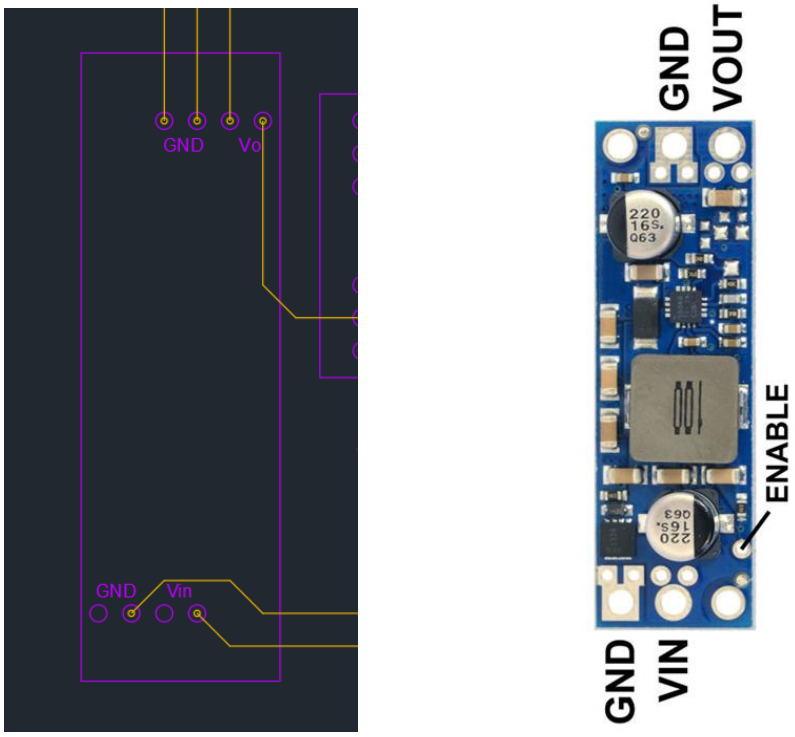



*figura 61: Step-Up Voltage Regulator U3V50F12*

Principalmente, el objetivo de este dispositivo consiste en la alimentación de las células de carga, alimentadas a 12 Voltios desde las bornas principales.

Gracias a este aumento del voltaje en la alimentación de los equipos de medida, es posible obtener con mayor resolución el peso detectado por las galgas extensiométricas instaladas en el interior de las células.

Esto se debe a que la sensibilidad de dichas galgas, dispuestas como un puente de Wheatstone, son el resultado de un factor dado por el fabricante multiplicado por la tensión entre los extremos del puente. Por este motivo, cuanto mayor sea el voltaje mejor resultará la medida.

Adicionalmente, para una mayor fiabilidad, se instala un inductor entre la salida del amplificador de tensión y las bornas principales de las galgas, atenuando posibles interferencias acopladas en el circuito.

Además de esto, desde el tren de 12Vcc se alimenta tanto el grupo de conmutadores como la primera línea de amplificadores operacionales, los cuales se emplean para aislar la señal recibida por las galgas extensiométricas.

Por último, este nivel de tensión también se emplea para la alimentación del generador de tensión aislada.

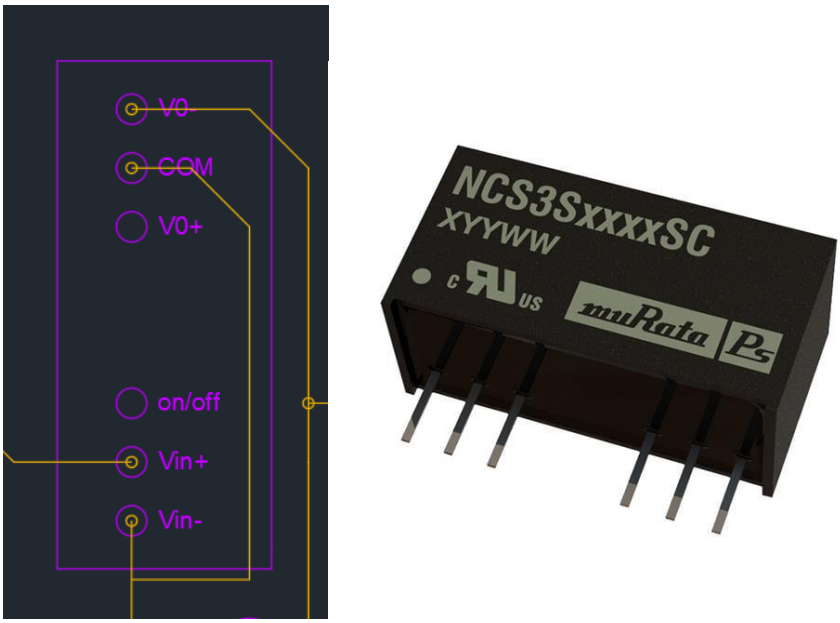



*figura 62: DC/DC Converter Isolated +/-3.3V 2W HS 85044095*

Este dispositivo, una vez alimentado por 12Vcc en los puertos de entrada, es capaz de generar una salida de voltaje de baja potencia equivalente a 3.3 Voltios positivos en su puerto 4 y 3.3 Voltios negativos en el puerto 6 (con respecto al común COM ubicado en el puerto 5).

Estos niveles de tensión, sin embargo, se encuentran aislados del resto del circuito de potencia. Es decir, es necesario proporcionarle una referencia desde donde se pueda obtener un nivel de voltaje estable.

En este caso, el dispositivo conversor de tensión aislada se ha instalado con el objetivo de obtener un nivel de tensión negativa en el circuito. Por este motivo, el puerto empleado para anclar al circuito ha sido el de 3.3V positivos, el cual se conecta a la tierra común del panel.

De esta manera, se obtiene a la salida del puerto 6 un voltaje de 6.6 voltios negativos.

Aunque resulte redundante y poco eficiente alimentar el conversor de tensión aislada desde la salida del amplificador *boost*, esto se ha hecho debido a la ausencia de una alternativa que pueda alimentarse desde 5Vcc, la cual cuenta con una fecha de reposición por el fabricante superior a los 6 meses.

#### *3.3.1.3: Bornas*

Para unir las células de carga al dispositivo, se emplea un grupo de bornas que permitan la fácil instalación de los equipos.

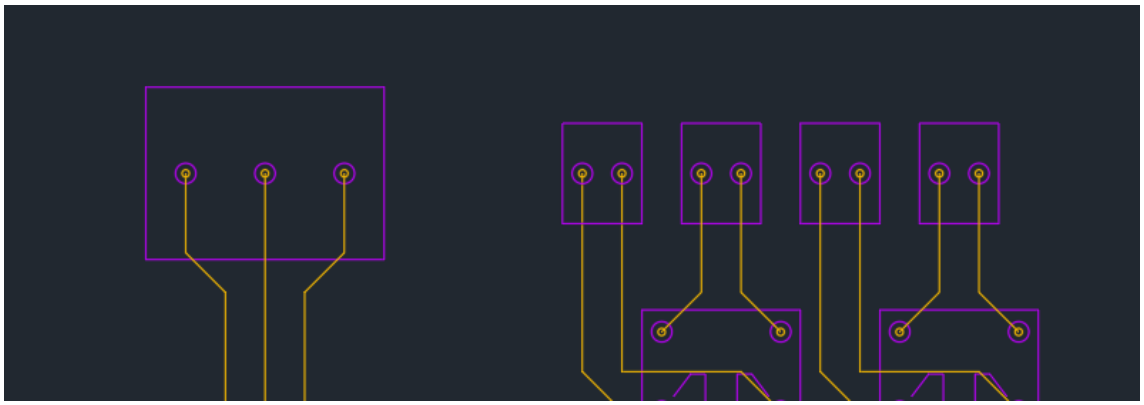
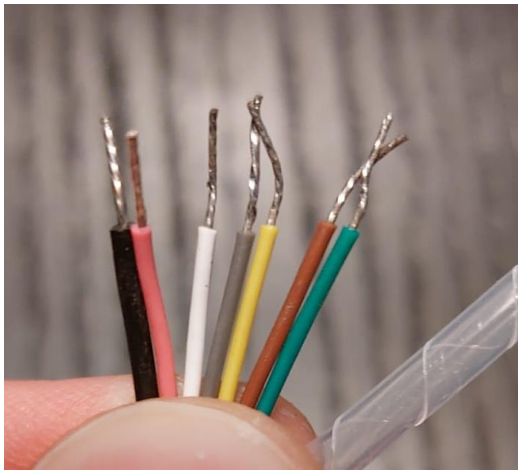


*figura 63: bornas*

Las células de carga empleadas cuentan con 7 puntas de cable en sus extremos, para la instalación de los equipos se debe realizar la siguiente conexión:



- Marrón -> 12Vcc
- Verde -> GND
- Amarillo -> señal +
- Blanco -> señal –
- Gris -> sensor +
- Rosa -> sensor –
- Negro -> Armadura (GND)

*figura 64: Cableado de las células de carga*

El motivo de la redundancia existente entre señal ± y sensor ± es debido al empleo de una doble conducción desde los extremos del puente de Wheatstone. Esta doble señal puede ser empleada por dispositivos más sofisticados que el del presente proyecto para contrarrestar la acción producida por la resistencia de los conductores. De esta forma se obtienen medidas más fiables. En el caso del montaje realizado para este proyecto, los extremos del sensor + (gris) y sensor – (rosa) han sido unidos a las bornas de 12Vcc y tierra junto con los extremos marrón y verde respectivamente.

De esta forma, aunque no de manera muy sofisticada, se logra reducir los efectos de la resistencia del conductor a la mitad.

Por lo tanto, la conexión resultante es la siguiente:

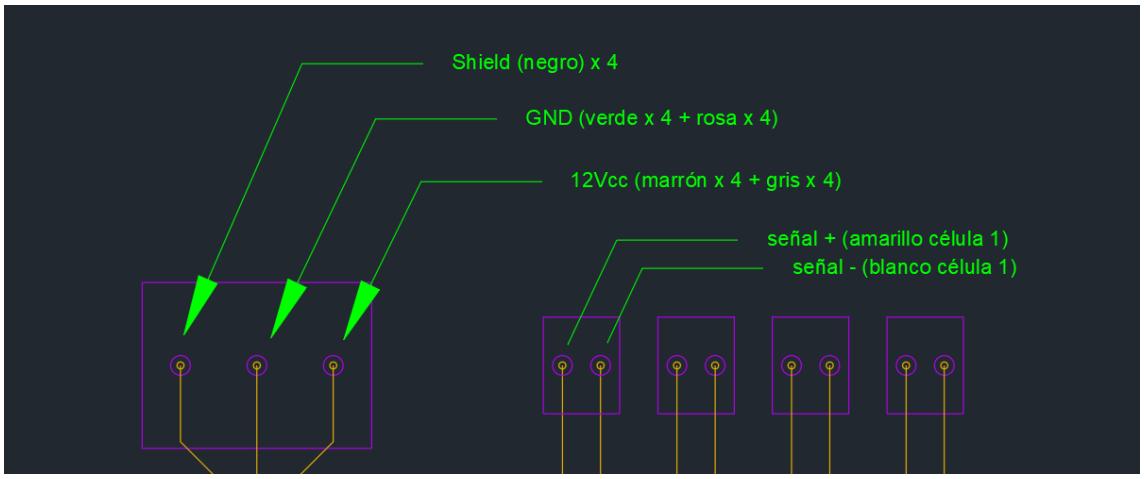



*figura 65: esquema de conexión de las células de carga al equipo*

El esquema de conexión de la señal que se muestra en la figura 65 se repite en las sucesivas bornas destinadas para el resto de las células de carga.

### *3.3.1.4: conmutadores de señal*

Debido a que el microcontrolador empleado para el desarrollo del dispositivo nivelador consta únicamente de tres puertos con capacidad de medida analógica, se decidió emplear simultáneamente los puertos ADC1 y ADC2 para la medida de las 4 células de peso.

Para ello se hace uso de dos relés conmutadores de 8 pines.

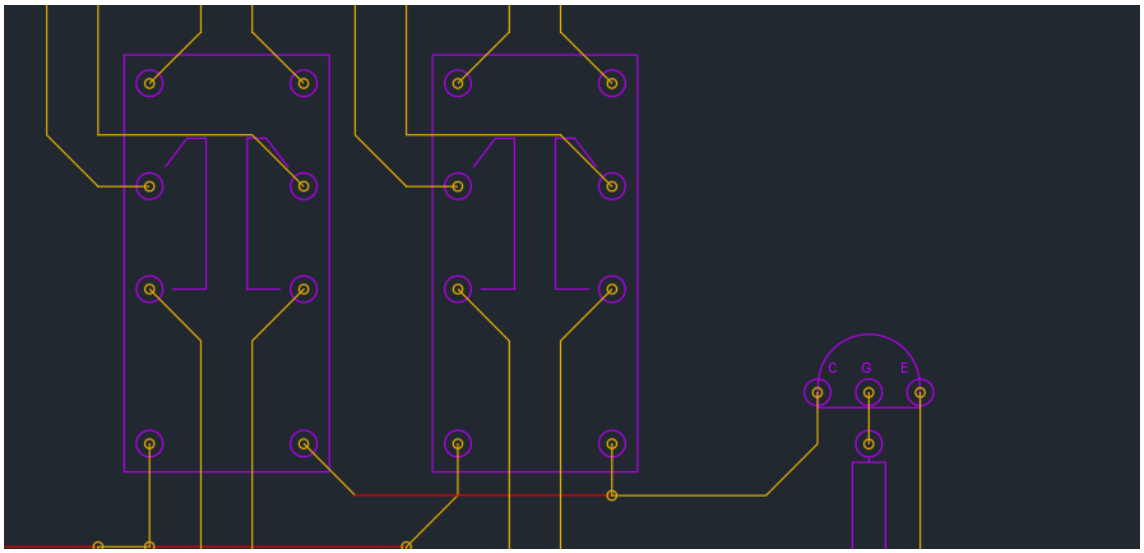



*figura 66: grupo de relés conmutadores de señal*

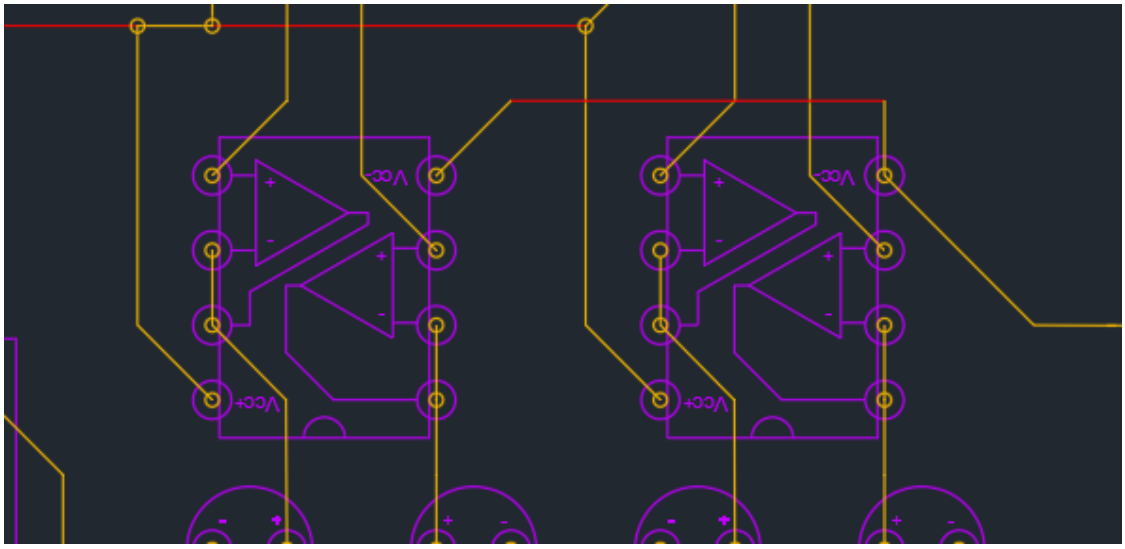
Estos relés conmutan ante la presencia de 12Vcc proporcionados por el elevador, a su vez esta actuación se ve controlada por la Raspberry a través de la acción de un transistor NPN conectado a la salida de ambos devanados.

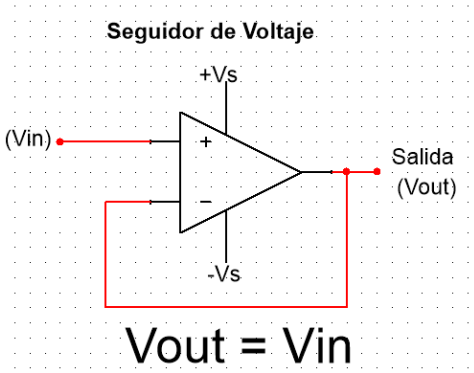
Mediante la actuación de estos conmutadores, es capaz de alternarse la lectura de las células de carga 1 / 3 en estado de reposo 2 / 4 en estado de excitación de los devanados.

### 3.3.1.5: Amplificador TL082

La función de la primera etapa del circuito de amplificación consiste en la separación de este del circuito de células de peso.

Para hacer efectiva esta separación, se dispone de un grupo de amplificadores operacionales (TL082) en configuración de "seguidor de voltaje"





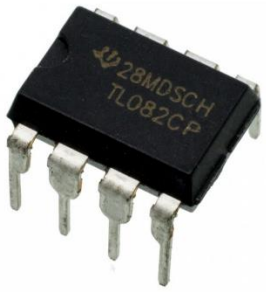




*figura 67: Amplificador operacional TL082 y configuración*

De esta manera, a la salida de cada amplificador operacional, se obtienen dos medidas virtualmente idénticas a aquellas que proporciona el puente de Wheatstone en sendos puntos de medida.

Esto permite trabajar con el voltaje de medida sin que este se pueda alterar en el proceso.

Al contar el dispositivo TL082 con dos circuitos de amplificadores operacionales, se emplea un único circuito integrado para aislar la pareja de señales generada por cada célula. El esquema de diseño, por lo tanto, está dispuesto de tal manera que cada conmutador proporciona ambas señales de trabajo pertenecientes a cada célula de carga al amplificador operacional.

Esta fase está seguida por una etapa de filtrado, efectuada por la disposición de cuatro condensadores electrolíticos dispuestos a la salida de cada circuito de amplificación.

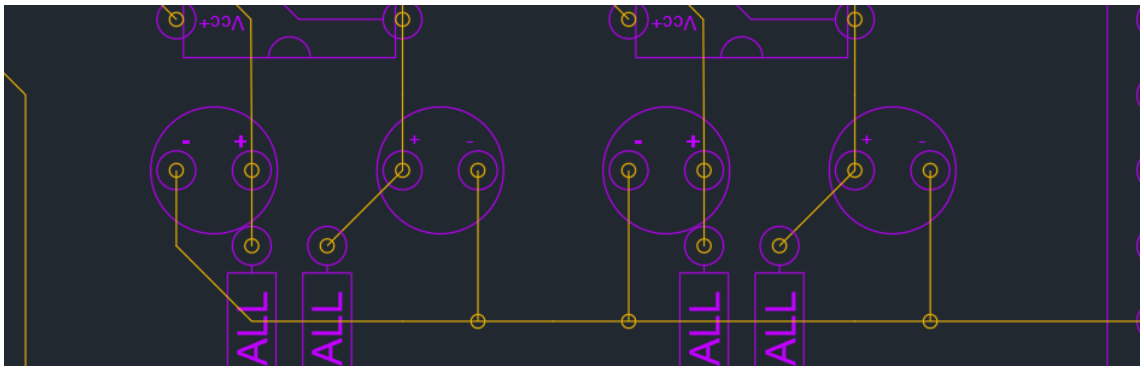



*figura 68: aislamiento de señal, etapa de filtrado*

### 3.3.1.6: Amplificador diferencial

Las células de peso empleadas para el prototipo del dispositivo de venta consisten en un grupo de 4 medidores de esfuerzo seccionante.

Estas consisten en la implementación de galgas extensiométricas en el interior del bloque mecanizado de acero, las cuales miden las torsiones sufridas por el material debido a los esfuerzos aplicados sobre este.

La medida de dicho esfuerzo se realiza entre los "centros" del puente de Wheatstone, conformado por la distribución de resistencias sensibles a la dilatación del material al que han sido adheridas.

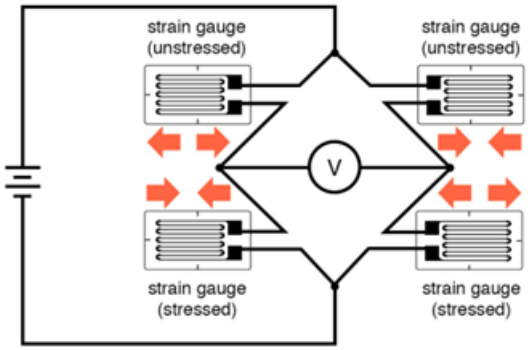



*figura 69: puente completo de Wheatstone*

En el caso de las células adquiridas, la sensibilidad de esta medida oscila entorno a los 2mV/V.

Esta sensibilidad se interpreta de tal manera que las células, en el caso de estar sometidas a su carga nominal, devuelven una diferencia de voltaje entre sus centros equivalente a 2mV por cada Voltio de voltaje que es proporcionado a sus extremos.

Para obtener una medida fiable, primero se eleva el voltaje de alimentación a los sensores de peso hasta un valor cercano al máximo recomendado por el fabricante (12V).

Esto aporta, en teoría, una diferencia máxima de 24mV entre los conductores de medida ante de una carga de 500Kg por punto de apoyo. Sin embargo, ya que la instalación de los equipos se realiza con estos en vacío, se sabe que el máximo peso soportado por cada célula es de aproximadamente 50kg (el modelo C-1870 cuenta con un peso de 200 Kg).

Siendo un orden de magnitud inferior al máximo de la escala utilizada, se obtiene una diferencia efectiva de 2,4mV, presentándose cada conductor a un voltaje aproximado de 6V con respecto a la tierra común del circuito.

Es por este motivo que, para realizar una medida exitosa en el proceso de instalación, se requiere de un circuito conocido como “amplificador diferencial”.

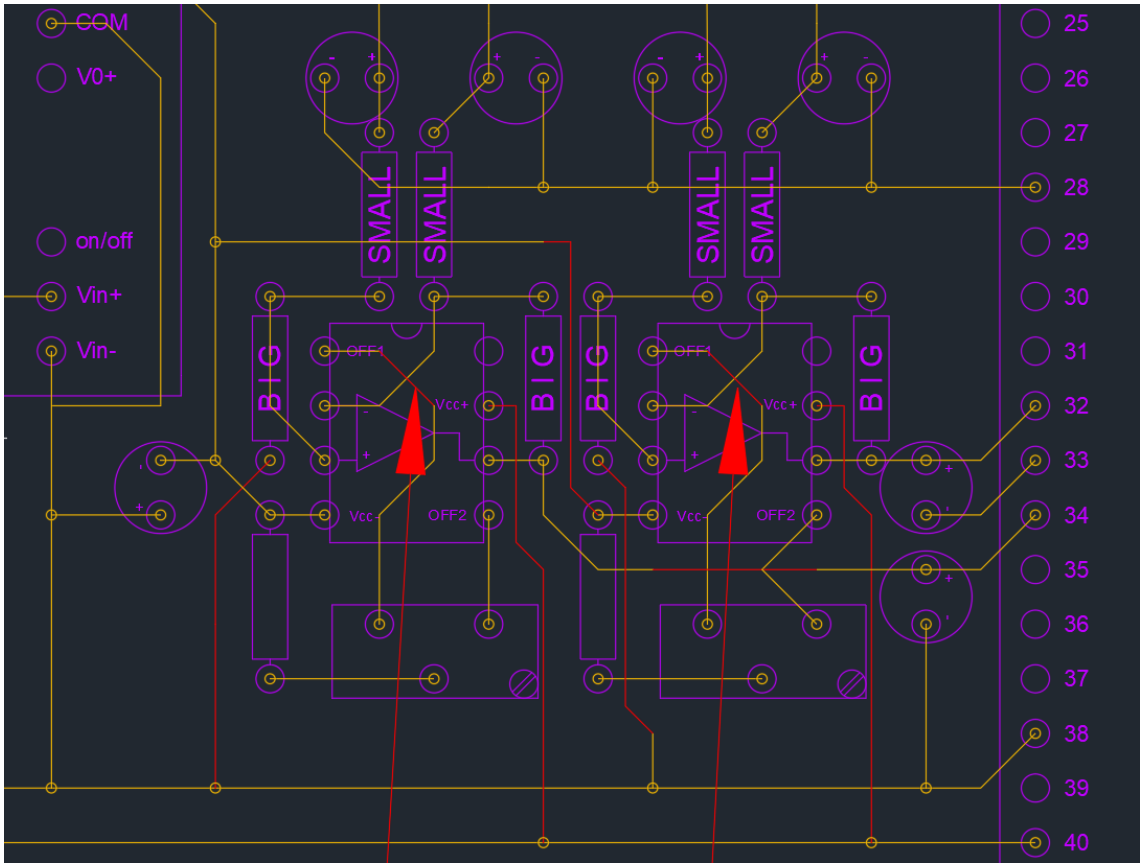



*figura 70: circuito de amplificación diferencial.*

Los circuitos integrados que han sido empleados para la realización de esta función son una pareja de amplificadores operacionales TL081, uno por cada célula, que se miden de manera simultánea.

Cada circuito integrado dispone de un amplificador operacional, a su vez, este cuenta con la posibilidad de introducir un *offset* a la señal de salida. Esta función es empleada para calibrar las medidas manualmente al inicio del proceso de instalación de los equipos.

El circuito mostrado en la figura 70 equivale al siguiente esquema:

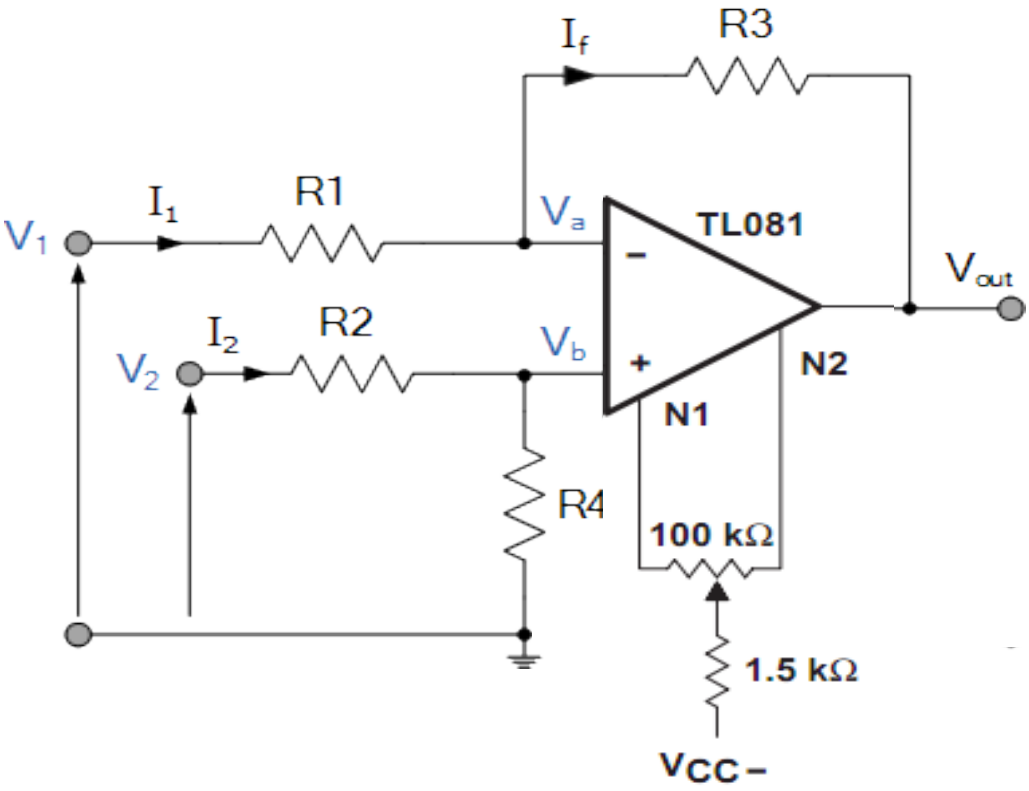



*figura 71: amplificador diferencial, esquema eléctrico*

De igualarse los valores de las resistencias R1 con R2 y R3 con R4, el resultado de Vout correspondería al siguiente:

$$V_{OUT} = \frac{R3}{R1} * (V_2 - V_1)$$

*Ecuación 1: Amplificador diferencial*

Los puertos N1 y N2 mostrados en la figura 71 se unen a los extremos de un potenciómetro, cuyo puerto común se conecta al nivel de voltaje negativo (necesario para su funcionamiento por lo tanto un valor de voltaje inferior a 0), a través de una resistencia de 1kΩ.

De esta manera, a través del potenciómetro, es posible ajustar el voltaje de salida entre sus extremos de saturación en estado de reposo para obtener un mayor rango de subida en la medida.

Los valores de las resistencias empleadas para R1 y R2 es de 1kΩ, mientras que los valores empleados para R3 y R4 equivalen a 1000k. Por lo tanto, la ganancia obtenida de este circuito es de alrededor de 1000 veces la diferencia de voltaje entre los terminales.

Esto supone un máximo en la medida de 2.4V. tres órdenes de magnitud superior a la original y significativamente más sencilla de medir con precisión.

### 3.3.2: Montaje final

El resultado final del circuito de balanceo puede apreciarse a continuación:

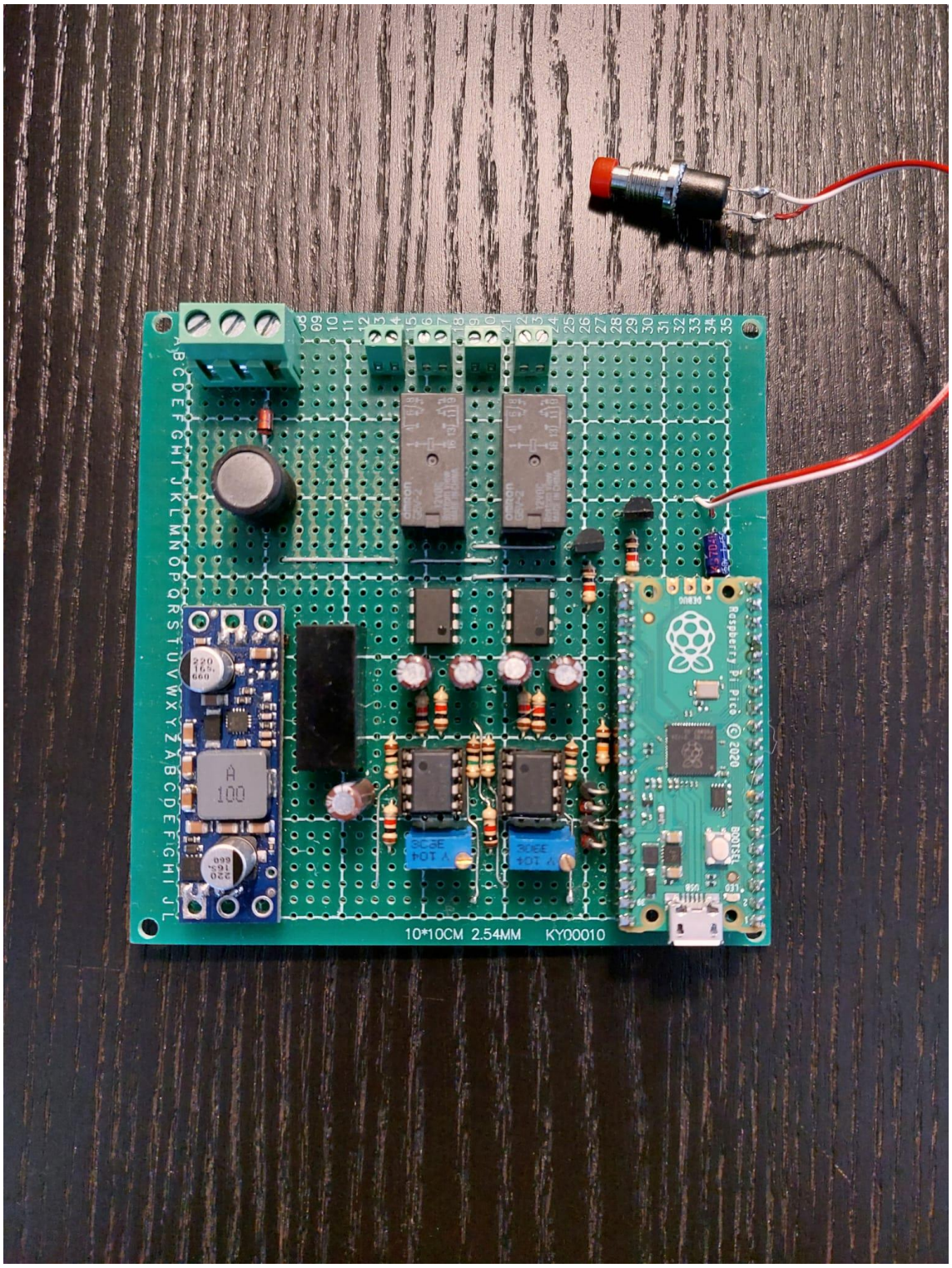



*figura 72: circuito final, vista frontal*

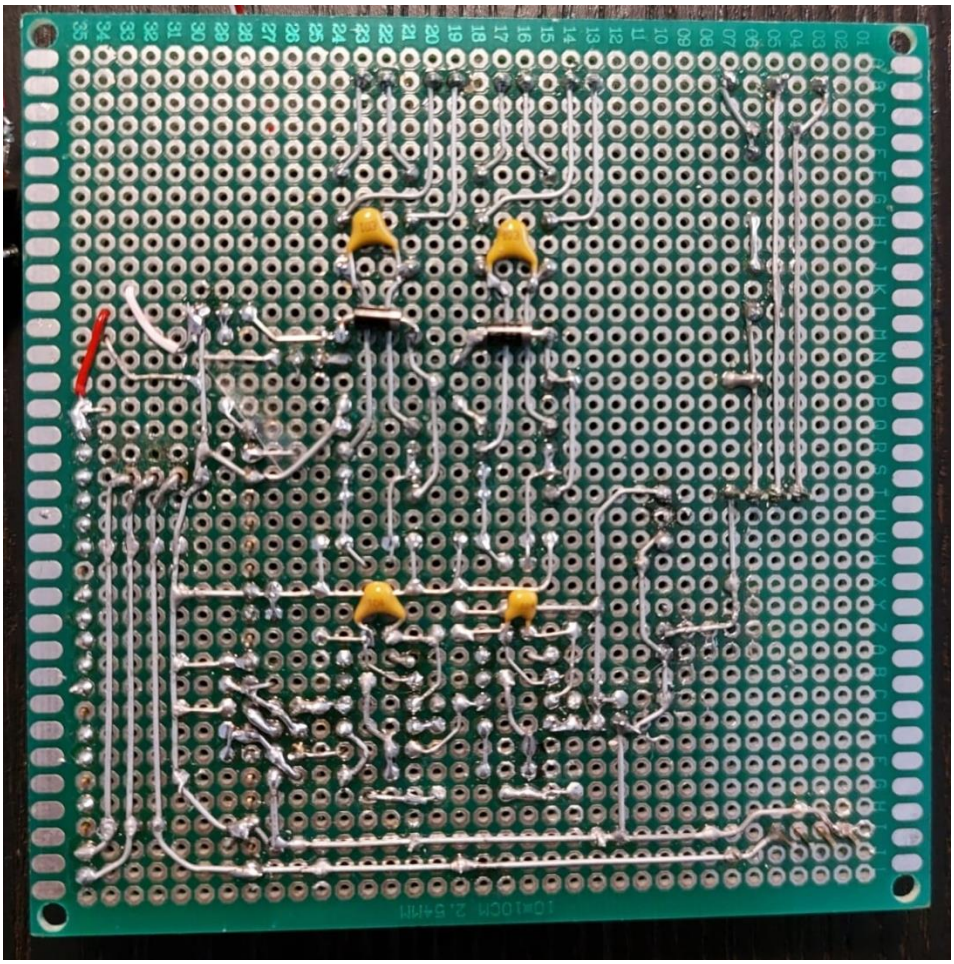


*figura 73: circuito final, vasta trasera*

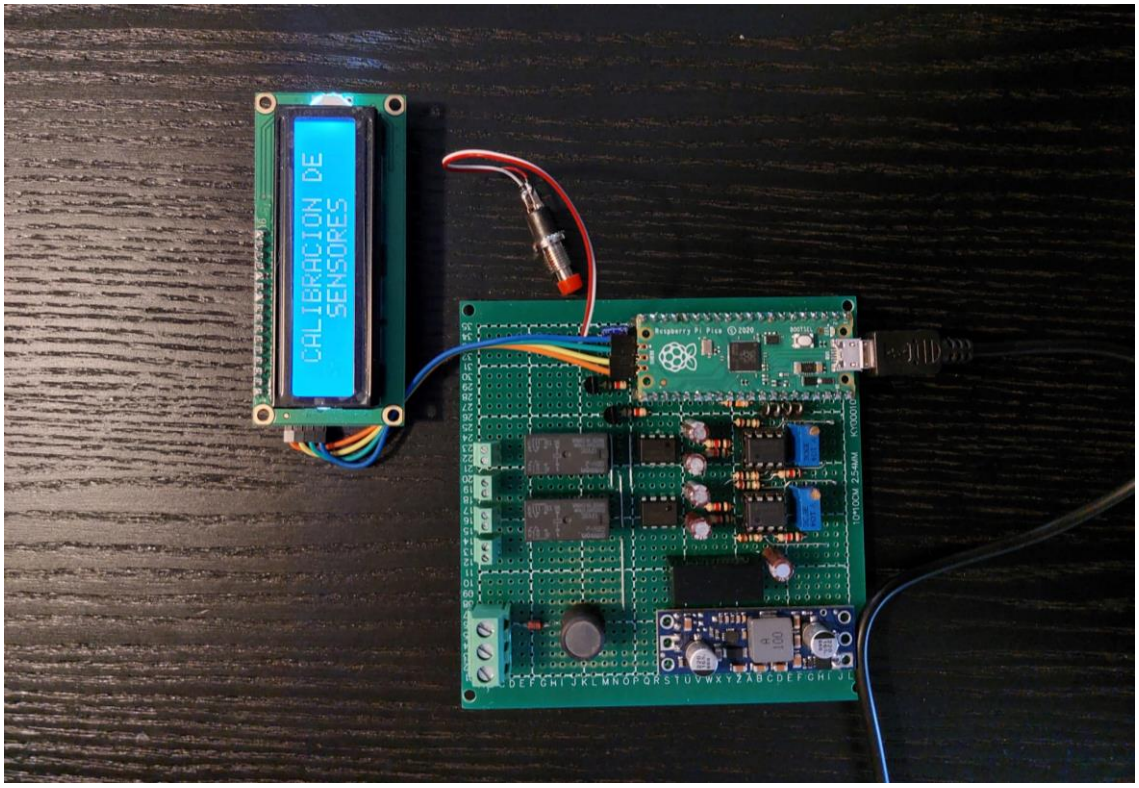



*figura 74: circuito final, montaje completo*

Adicionalmente, se realizó el diseño de un encapsulado que proteja el interior del circuito, al tiempo que facilita el acceso a los potenciómetros, bornas y el puerto de alimentación.


*figura 75: Encapsulado, vista frontal*

*figura 76: Encapsulado, detalle bornas*

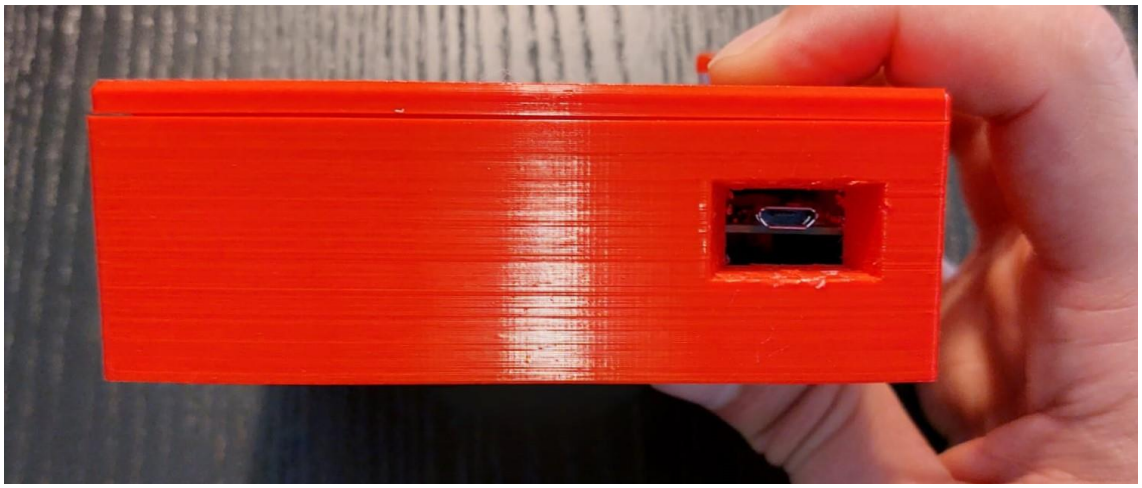


*figura 77: Encapsulado, detalle puerto de alimentación*

El código integrado en la Raspberry Pi Pico se expondrá en el Anexo II

### 3.3.3: Lecciones aprendidas

La producción del modelo final se encontró de forma inevitable con una serie de inconveniencias que dificultan el montaje del circuito.

A continuación, se desarrollan las lecciones aprendidas del montaje de dicho modelo:

#### *3.3.3.1: Voltaje negativo*

Durante los ensayos realizados sobre la *protoboard*, al intentar aplicar el circuito expuesto en la figura 71 (Ver 3.3.1), los resultados obtenidos del modelado del amplificador diferencial fueron mayoritariamente infructíferos.

Habiéndose aplicado todos los conocimientos adquiridos, el voltaje obtenido a la salida del amplificador operacional TL081 parecía no responder debidamente al *offset*, permaneciendo saturado en la mayor parte del rango del potenciómetro (cuyos extremos estaban puestos a Vcc y GND) y descendiendo ligeramente mediante un salto escalonado en el valor extremo de este.

Tras una investigación en mayor profundidad sobre las características del circuito integrado en el interior del amplificador, se llegó a la conclusión de la necesidad de un nivel de voltaje considerablemente negativo para la obtención de un valor de salida próximo a los 0V.

De no ser así, el *output* no sería capaz de aproximarse a dicho valor. De este descubrimiento surgió la necesidad de implementar un dispositivo capaz de proporcionar un nivel de voltaje negativo estable.

Durante la realización de estas pruebas, previo a la obtención del transformador de tensión aislado (ver punto 3.3.1.2), se hizo empleo de un regulador de tensión en línea de +6Vcc (L7806CV).

De esta forma, haciendo uso de esto dispositivo regulador de voltaje se obtiene el siguiente esquema y los niveles de voltaje indicados:

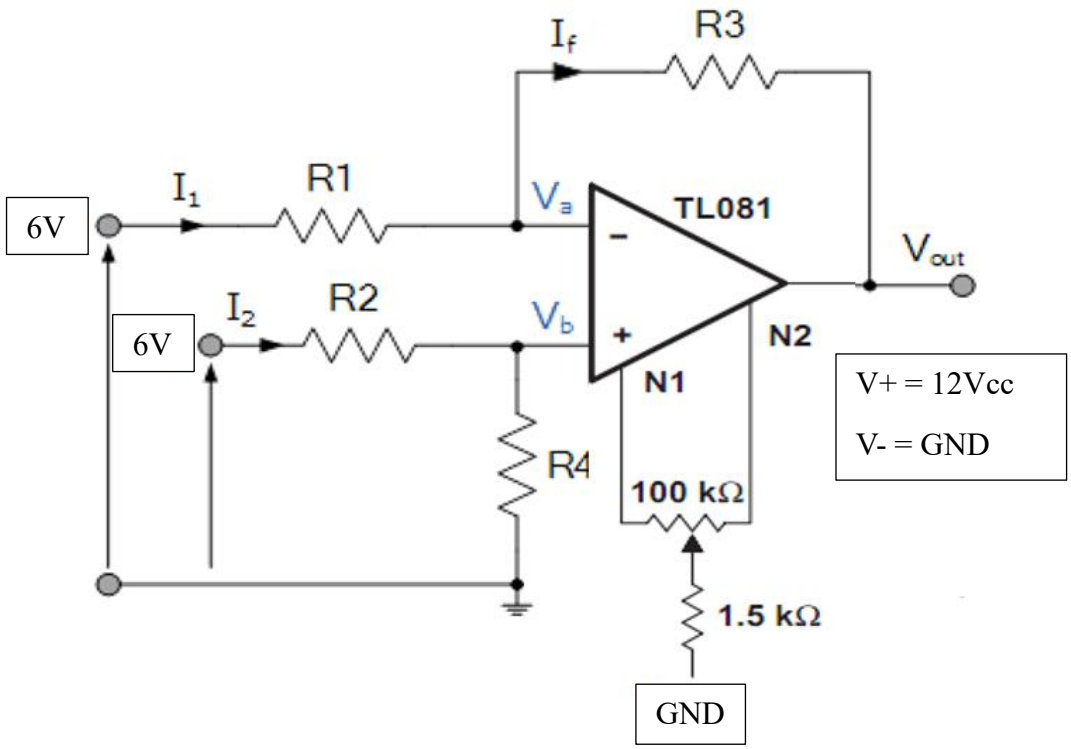



*figura 78: esquema de pruebas con voltajes negativos virtuales*

Haciendo uso de 6V en ambos puertos de entrada del amplificador diferencial, se puede simular este nivel de voltaje como un nivel base o tierra virtual. De esta manera, la alimentación del amplificador a 12V se comporta a efectos prácticos como un nivel de voltaje equivalente a 6Vcc con respecto a este. Mientras que la tierra conectada tanto a la alimentación del amplificador en V- como a la resistencia unida al común del potenciómetro, resulta en el equivalente de -6Vcc.

Con esta nueva distribución el circuito fue capaz de modular su salida (variando entorno a los 6Vcc), respondiendo a las modificaciones realizadas en el potenciómetro.

Ante este resultado, se realizó la compra de un dispositivo regulador de tensión aislado, permitiendo la generación de un nivel de voltaje negativo estable que pudiera hacer la función de V-.

### *3.3.3.2: Saturación del amplificador operacional*

Una vez finalizado el montaje de la placa, el resultado de voltaje obtenido a la salida del amplificador operacional no correspondía al mismo obtenido durante las pruebas realizadas en la *protoboard*.

Al conectar la célula de peso al dispositivo, el circuito de amplificación permanecería saturado, sin llegar a mostrar ninguna respuesta significativa a las modificaciones del *offset* a través del potenciómetro.

Al no encontrar ninguna solución, se procedió al estudio del circuito integrado del amplificador TL081.

**schematic (each amplifier)**

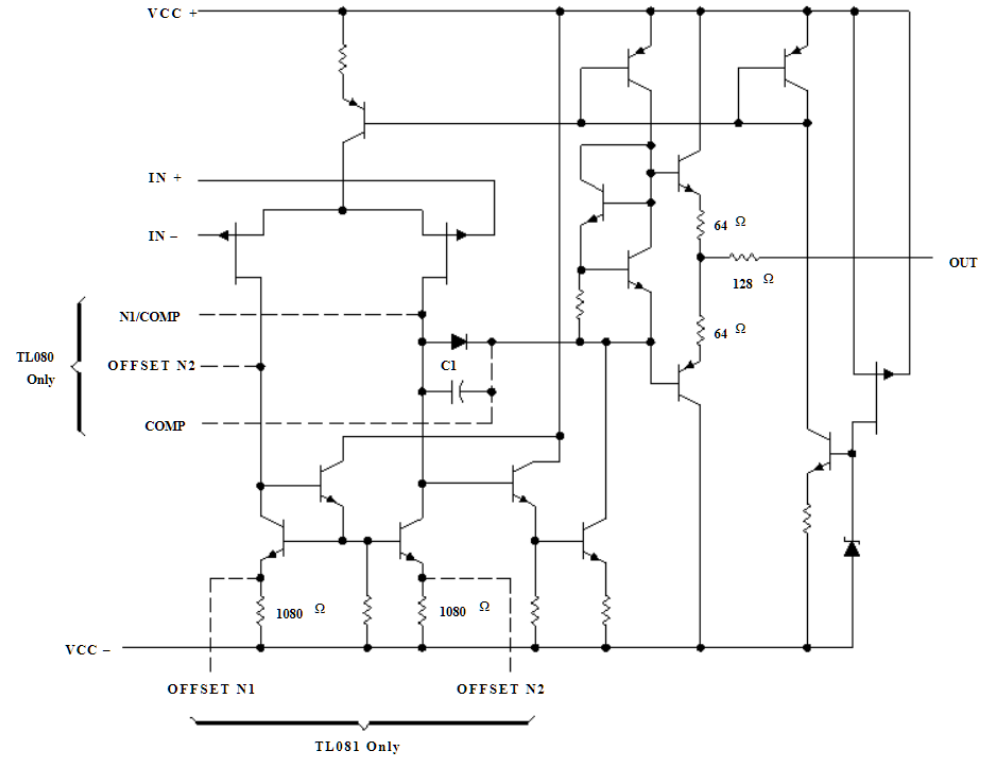



*figura 79: TL081 Datasheet, Texas Instruments*

Tras un estudio en profundidad del esquema funcional representado en el documento *datasheet* de *Texas Instrments* [TEXA92], se llegó a la conclusión de que el origen del problema estaba siendo generado en los puertos de entrada al amplificador *IN+ IN-*.

Estos puertos se unen al circuito a través de la puerta (*gate*) de un transistor JFET de canal P

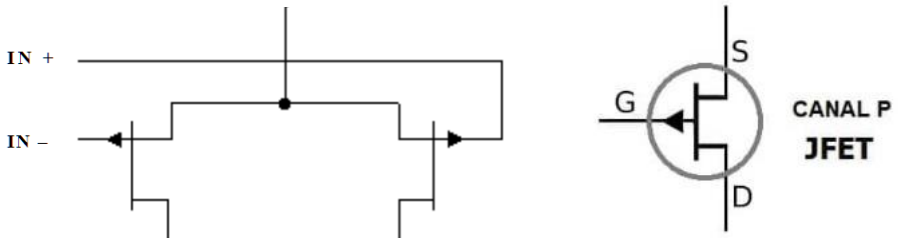



*figura 80: Vista en detalle del CI TL081*

Estos transistores JFET, a su vez, tienen acoplados en su fuente (*source*) el puerto Vcc+ a través del colector de un transistor tipo PNP.

El problema generado en el circuito se debe a la diferencia de voltaje entre la alimentación del amplificador operacional (V+) y las señales de medida proporcionadas por las células de peso (N1 y N2).

Al no haberse considerado necesario un voltaje de salida del circuito de amplificación diferencial superior al que el microcontrolador es capaz de medir, se conectó el puerto V+ de los amplificadores TL081 al tren de 5Vcc.

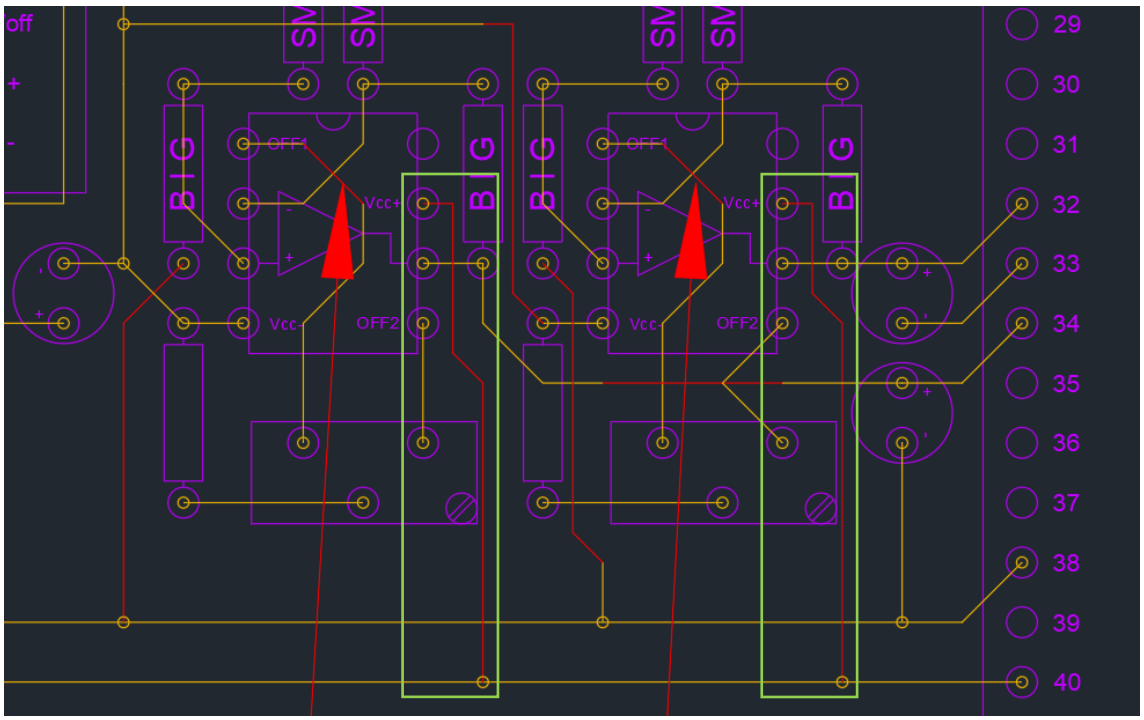



*figura 81: Alimentación de los amplificadores TL081*

Esto impide que, en el interior del amplificador operacional, conmuten los transistores JFET, al presentar un mayor nivel de voltaje en la puerta (*gate*) que en la fuente (*source*). Provocando, por lo tanto, el mal funcionamiento del circuito y generando una saturación constante del mismo.

Con el fin de solucionar este problema basta con modificar la alimentación de los amplificadores operacionales, conectándolos al tren de 12Vcc a la salida del elevador de tensión.

Sin embargo, con tal de evitar posibles saturaciones de un valor innecesariamente elevado que puedan dañar el microcontrolador, se optó por modificar ligeramente la tensión de alimentación de las células de peso.

Para ello se instaló a la salida del amplificador de tensión un diodo Zener que reduce mínimamente la tensión en bornas de salida por 3,3V.

Mediante esta modificación, el voltaje devuelto por el puente de Wheatstone en sus centros se redujo de 6V a 4,35V, permitiendo la conmutación de los transistores JFET y el correcto funcionamiento del circuito de amplificación diferencial.

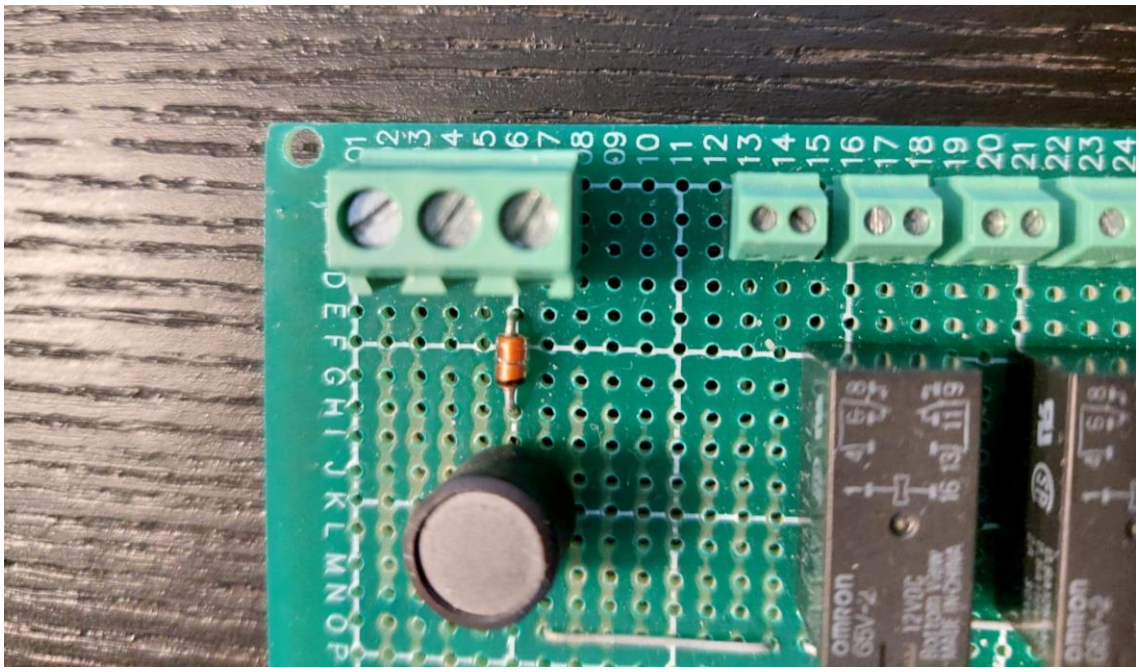



*figura 82: circuito de alimentación de galgas, vista en detalle del diodo Zener*

### *3.3.3.3: Niveles de voltaje, puertos ADC*

Durante las pruebas del circuito de amplificación diferencial (punto 3.3.3.2), la Raspberry Pi Pico permaneció conectada a la placa para suministrar el voltaje necesario y la unión de los puertos de tierra realizados a sus respectivos pines por el resto de los componentes.

Durante dichas pruebas, los puertos ADC1 y ADC2 del microcontrolador permanecieron expuestos a un voltaje próximo a los 5Vcc. Esto pasó desapercibido hasta que posteriormente, los amplificadores comenzaron a fallar definitivamente, sin mostrar respuesta a los voltajes mostrados a la entrada del circuito de amplificación.

Tras una observación más minuciosa, se comprobó que estos amplificadores se habían dañado al permanecer sus salidas saturadas a su valor máximo y cortocircuitadas a tierra durante largos periodos de tiempo.

Esto se debía a que los puertos ADC1 y ADC2, ahora permanecían cortocircuitados a tierra, dando continuidad entre ellos y los puertos GND del microcontrolador.

Tras una extensa lectura del manual de fabricante de la Raspberry [RASP23], observamos la siguiente nota en el apartado 2.9.5:

### 2.9.5. ADC Supply (ADC_AVDD)

ADC_AVDD supplies the chip's Analogue to Digital Converter (ADC). It can be powered at a nominal voltage between 1.8V and 3.3V, but the performance of the ADC will be compromised at voltages below 2.97V. To reduce the number of external power supplies, ADC_AVDD can use from the same power source as the digital IO supply (IOVDD).

**NOTE**

It is safe to supply ADC_AVDD at a higher or lower voltage than IOVDD, e.g. to power the ADC at 3.3V, for optimum performance, while supporting 1.8V signal levels on the digital IO. But the voltage on the ADC analogue inputs must not exceed IOVDD, e.g. if IOVDD is powered at 1.8V, the voltage on the ADC inputs should be limited to 1.8V. Voltages greater than IOVDD will result in leakage currents through the ESD protection diodes. See Section 5.5.3, "Pin Specifications" for details.

*figura 83: indicaciones de fabricante, Raspberry Pi Pico*

donde IOVDD representa el voltaje empleado en los puertos digitales, la medida en los puertos digitales de un voltaje superior al empleado en el propio microcontrolador supondrá la aparición de corrientes parásitas dentro del circuito de medida que finalmente dañarán el sistema. En este caso produciendo un cortocircuito a tierra en el interior del microcontrolador.

Tras la sustitución del microcontrolador dañado, y con tal de evitar que se repitiera este error, se introdujo a la salida del circuito diferencial el siguiente esquema de componentes:

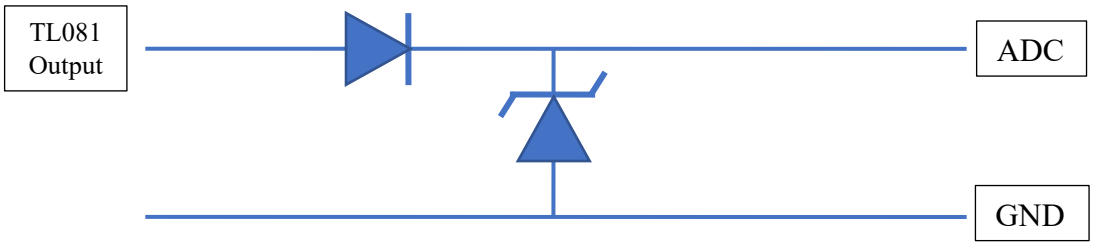



*figura 84: circuito de protección analógica*

De esta manera, se protege el circuito de medida interno de la Raspberry de sobretensiones (gracias al diodo Zener de 3,3V) y de voltajes negativos (gracias al diodo).

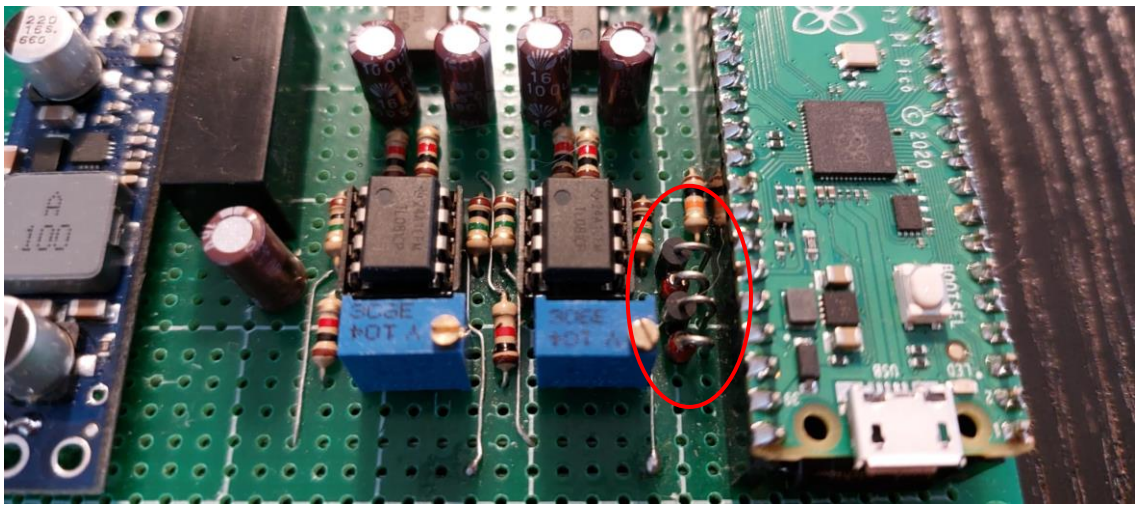


*figura 85: circuito de protección ADC, vista en detalle*

### *3.3.3.4: estabilidad de voltaje*

Durante los ensayos finales del nivelador, se pudo observar que la actuación de los conmutadores de señal (ver 3.3.1.4), alimentados desde el tren de 12Vcc, alteraba significativamente el voltaje de salida del amplificador de tensión al aumentar el consumo de este.

Esto provocaba un voltaje inferior en segundo grupo de células de carga medido, generando un conflicto en la fiabilidad del voltaje.

Para resolver este problema se propuso la división de la actuación de los conmutadores, de tal forma que durante la primera medida el conmutador del primer grupo se activara mientras que el conmutador secundario permaneciera en estado de reposo, estos estados se intercambiarían al realizar la segunda medida.

De esta forma, uno de los conmutadores permanecería en todo momento activado, equilibrando la carga del amplificador de tensión.

Para esto se incluyó un segundo transistor, separando la salida de los conmutadores y derivando cada una a su respectivo interruptor.

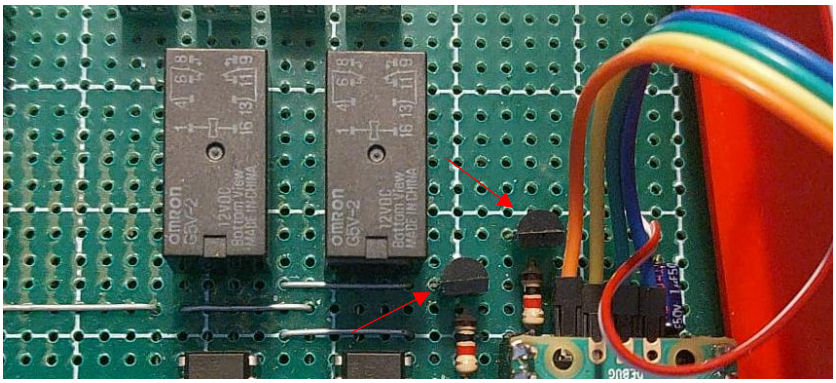



*figura 86: vista en detalle, transistores y conmutadores*

Adicionalmente se incluyen diodos en las bobinas de los conmutadores para evitar picos de voltaje debido a su interrupción:

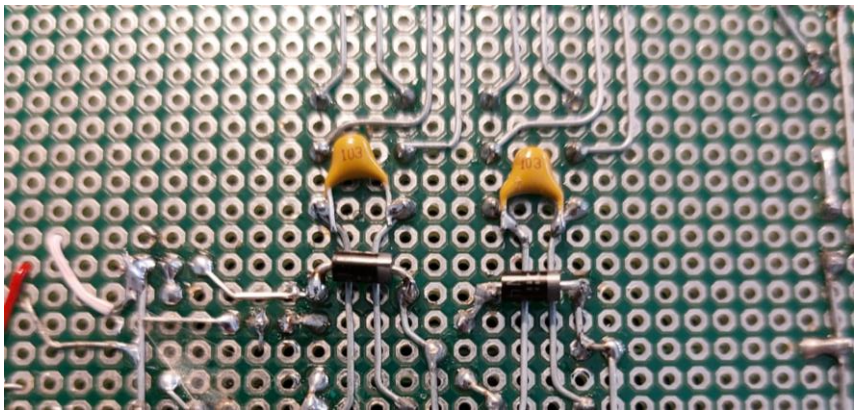


*figura 87: vista en detalle, diodos en la bobina de los conmutadores*

## ANEXO I.

El presente proyecto se alinea con los siguientes objetivos de desarrollo sostenible:


Industria, innovación e infraestructura: El objetivo principal del proyecto es la fabricación y distribución de una serie de dispositivos accesibles para el público que faciliten la obtención de un producto en concreto, generando de esta manera una red de almacenamiento y venta.


Trabajo decente y crecimiento económico: La distribución y puesta en marcha de estos dispositivos generarán una demanda laboral específica para cubrir los puestos de mantenimiento y reabastecimiento de los congeladores.

# ANEXO II



## >>> main.py

```
from machine import Pin, I2C, ADC
from esp8266_i2c_lcd import I2cLcd
import time
from time import ticks_ms
import funciones as fun


DEFAULT_I2C_ADDR = 0x27

i2c = I2C(0, scl=Pin(1), sda=Pin(0), freq=400000)
lcd = I2cLcd(i2c, DEFAULT_I2C_ADDR, 2, 16)
time.sleep(1/200)

##################################

relay1 = Pin(16,Pin.OUT)
relay1.value(0)
relay2 = Pin(17,Pin.OUT)
relay2.value(0)
puls = Pin(15,Pin.IN, Pin.PULL_UP)
zero = fun.mide(0)
zero = fun.media(zero,0)
est = True
cel = 1
pulsFlag = False
lastPuls = 0

"""comprobación de sensores para evitar daños"""
lcd.clear()
sat2 = fun.media(fun.mide(2),0)
sat1 = fun.media(fun.mide(1),0)
if(sat1 == "SAT"):
    lcd.move_to(0,0)
    lcd.putstr("SENSOR IZQ SAT")
if(sat2 == "SAT"):
    lcd.move_to(0,1)
    lcd.putstr("SENSOR DER SAT")
if(sat1 == "SAT" or sat2 == "SAT"):
    time.sleep(2)
    lcd.clear()
    lcd.move_to(0,0)
    lcd.putstr("apague y gire en")
    lcd.move_to(0,1)
    lcd.putstr("sentido horario.")
    while True:
        pass

"""creamos la funcion de interurpcion"""
def callback(puls):
```

```
    global cel, est, pulsFlag, lastPuls
    if (time.ticks_ms()-lastPuls)>500:
        dt = 0
        t0 = time.ticks_ms()
        while puls.value() == 0 and dt <= 2000:
            time.sleep(20/1000)
            dt = time.ticks_ms() - t0
        lcd.clear() #limpiamos pantalla para que se sepa de que se ha recibido una señal
        if dt >= 2000: #pulsacion larga (cambio de estado calib <=> medida)
            est = not est
        else:
            if est == True: #pulsacion corta y estamos en calib
                cel = cel + 1
                if cel > 4:
                    cel = 1
            else: #pulsacion corta y estamos en med
                pulsFlag = True
    lastPuls = time.ticks_ms()
puls.irq(trigger=puls.IRQ_FALLING, handler = callback)



while True:

    """esta parte del codigo es para la visualización y balanceado de galgas"""
    if est == True: #True == calibración
        lcd.clear()
        lcd.move_to(1,0)
        lcd.putstr("CALIBRACION DE")
        lcd.move_to(4,1)
        lcd.putstr("SENSORES")
        time.sleep(2)
        lcd.clear()
        cel = 1
        while est:
            if cel == 1:
                sens = 2
                relay1.value(0)
                relay2.value(1)
            if cel == 2:
                sens = 2
                relay1.value(1)
                relay2.value(0)
            if cel == 3:
                sens = 1
                relay1.value(0)
                relay2.value(1)
            if cel == 4:
                sens = 1
                relay1.value(1)
                relay2.value(0)

            time.sleep(0.5)
            vec = fun.mide(sens)
            med = fun.media(vec,zero)
            fmax = fun.MAX(vec)
            fmin = fun.MIN(vec)
            lcd.clear()
            lcd.move_to(0,0)
            lcd.putstr("cel %d"%cel)
            lcd.move_to(6,0)
            if(med == "SAT"):
                lcd.putstr("med: SAT")
```

```
        elif(med == "NEG"):
            lcd.putstr("med: NEG")
        else:
            lcd.putstr("med: %d"%med)
        lcd.move_to(0,1)
        lcd.putstr("m:%d"%fmin)
        lcd.move_to(9,1)
        lcd.putstr("M:%d"%fmax)

    """a partir de aquí empieza el código de medida"""
    if not est: #False == medida
        lcd.clear()
        lcd.move_to(3,0)
        lcd.putstr("BALANCEADO")
        lcd.move_to(3,1)
        lcd.putstr("DE CELULAS")
        time.sleep(2)
        lcd.clear()

        while not est:
            lcd.clear()
            time.sleep(0.5)
            lcd.move_to(1,0)
            lcd.putstr("retire el peso")
            lcd.move_to(0,1)
            lcd.putstr("y pulse el boton")
            pulsFlag = False #just in case le pega un rebote al botón
            caralho = False #caralho es un indicador de que algo no fue gucci

            while not pulsFlag and not est:
                time.sleep(1/1000)
            pulsFlag = False
            """realizamos la medida de los ceros"""
            if not est:
                lcd.clear()
                lcd.move_to(0,0)
                lcd.putstr("midiendo ref...")

                relay1.value(0)
                relay2.value(1)
                time.sleep(0.5)
                lcd.move_to(0,1)
                lcd.putstr("1")
                z1 = fun.media(fun.mide(2),zero) #la primera galga (empezando por la derecha) comunica con el operacional
derecho que está en el ADC2
                time.sleep(0.5)
                lcd.move_to(0,1)
                lcd.putstr("2")
                z3 = fun.media(fun.mide(1),zero) #la tercera galga (empezando por la derecha) comunica con el operacional
izquierdo que está en el ADC1
                relay1.value(1)
                relay2.value(0)
                time.sleep(0.5)
                lcd.move_to(0,1)
                lcd.putstr("3")
                z2 = fun.media(fun.mide(2),zero)
                time.sleep(0.5)
                lcd.move_to(0,1)
                lcd.putstr("4")
                z4 = fun.media(fun.mide(1),zero)
                relay1.value(0)
                relay2.value(1)
```

```
z=[z1,z2,z3,z4] #ordenamos el vector

"""comprobamos que no haya saturado ninguna célula o lo que sea por lo que sea"""
for i in range(4):
    if z[i] == "SAT":
        lcd.clear()
        lcd.move_to(0,0)
        num = i+1
        lcd.putstr("cel %d saturada"%num)
        lcd.move_to(0,1)
        lcd.putstr("pulse para cont")
        caralho = True
        while not pulsFlag and not est:
            time.sleep(1/1000)
        pulsFlag = False
    if z[i] == "NEG":
        lcd.clear()
        lcd.move_to(0,0)
        num = i+1
        lcd.putstr("cel %d negativa"%num)
        lcd.move_to(0,1)
        lcd.putstr("pulse para cont")
        caralho = True
        while not pulsFlag and not est:
            time.sleep(1/1000)
        pulsFlag = False
time.sleep(0.5)
if not caralho:
    lcd.clear()
    lcd.move_to(0,0)
    lcd.putstr("apoye la maquina")
    lcd.move_to(0,1)
    lcd.putstr("y pulse el boton")
    while not pulsFlag and not est:
        time.sleep(1/1000)
    pulsFlag = False
    if not est:
        lcd.clear()
        lcd.move_to(0,0)
        lcd.putstr("midiendo cel...")

        relay1.value(0)
        relay2.value(1)
        time.sleep(0.5)
        lcd.move_to(0,1)
        lcd.putstr("1")
        m1 = fun.media(fun.mide(2),zero)
        time.sleep(0.5)
        lcd.move_to(0,1)
        lcd.putstr("2")
        m3 = fun.media(fun.mide(1),zero)
        relay1.value(1)
        relay2.value(0)
        time.sleep(0.5)
        lcd.move_to(0,1)
        lcd.putstr("3")
        m2 = fun.media(fun.mide(2),zero)
        time.sleep(0.5)
        lcd.move_to(0,1)
        lcd.putstr("4")
        m4 = fun.media(fun.mide(1),zero)
        relay1.value(0)
        relay2.value(1)
```

```
m=[m1,m2,m3,m4]

for i in range(4):
    if m[i] == "SAT":
        lcd.clear()
        lcd.move_to(0,0)
        num = i+1
        lcd.putstr("cel %d saturada"%num)
        lcd.move_to(0,1)
        lcd.putstr("pulse para cont")
        caralho = True
        while not pulsFlag and not est:
            time.sleep(1/1000)
        pulsFlag = False
    if m[i] < z[i]: #en caso de que mida menos que antes es que la galga está invertida
        lcd.clear()
        lcd.move_to(0,0)
        num = i+1
        lcd.putstr("cel %d invertida"%num)
        lcd.move_to(0,1)
        lcd.putstr("pulse para cont")
        caralho = True
        while not pulsFlag and not est:
            time.sleep(1/1000)
        pulsFlag = False
"""visualización de los resultados en %"""
if not caralho:
    d=[0,0,0,0]
    p=[0,0,0,0]
    for i in range(4):
        d[i]=m[i]-z[i]
    tot=d[0]+d[1]+d[2]+d[3]
    for i in range(4):
        p[i]=(d[i]/tot)*100
    lcd.clear()
    for i in range(4):
        if i == 0:
            lcd.move_to(0,0)
        if i == 1:
            lcd.move_to(9,0)
        if i == 2:
            lcd.move_to(0,1)
        if i == 3:
            lcd.move_to(9,1)
        num = i+1
        lcd.putstr("%d:"%num)
        lcd.putstr("%d"%p[i])
        lcd.putstr(",")
        r=(p[i]-int(p[i]))*100
        if r<10:
            lcd.putstr("0")
        lcd.putstr("%d"%r)
    time.sleep(1)
    while not pulsFlag and not est:
        time.sleep(1/1000)
    pulsFlag = False
```

## >>> esp8266_i2c_lcd.py [HYLA21]

```
"""Implements a HD44780 character LCD connected via PCF8574 on I2C.
  This was tested with: https://www.wemos.cc/product/d1-mini.html"""

from lcd_api import LcdApi
from time import sleep_ms

# The PCF8574 has a jumper selectable address: 0x20 - 0x27
DEFAULT_I2C_ADDR = 0x27

# Defines shifts or masks for the various LCD line attached to the PCF8574

MASK_RS = 0x01
MASK_RW = 0x02
MASK_E = 0x04
SHIFT_BACKLIGHT = 3
SHIFT_DATA = 4


class I2cLcd(LcdApi):
  """Implements a HD44780 character LCD connected via PCF8574 on I2C."""

  def __init__(self, i2c, i2c_addr, num_lines, num_columns):
    self.i2c = i2c
    self.i2c_addr = i2c_addr
    self.i2c.writeto(self.i2c_addr, bytearray([0]))
    sleep_ms(20)  # Allow LCD time to powerup
    # Send reset 3 times
    self.hal_write_init_nibble(self.LCD_FUNCTION_RESET)
    sleep_ms(5)  # need to delay at least 4.1 msec
    self.hal_write_init_nibble(self.LCD_FUNCTION_RESET)
    sleep_ms(1)
    self.hal_write_init_nibble(self.LCD_FUNCTION_RESET)
    sleep_ms(1)
    # Put LCD into 4 bit mode
    self.hal_write_init_nibble(self.LCD_FUNCTION)
    sleep_ms(1)
    LcdApi.__init__(self, num_lines, num_columns)
    cmd = self.LCD_FUNCTION
    if num_lines > 1:
      cmd |= self.LCD_FUNCTION_2LINES
    self.hal_write_command(cmd)

  def hal_write_init_nibble(self, nibble):
    """Writes an initialization nibble to the LCD.

    This particular function is only used during initialization.
    """
    byte = ((nibble >> 4) & 0x0f) << SHIFT_DATA
    self.i2c.writeto(self.i2c_addr, bytearray([byte | MASK_E]))
    self.i2c.writeto(self.i2c_addr, bytearray([byte]))

  def hal_backlight_on(self):
    """Allows the hal layer to turn the backlight on."""
    self.i2c.writeto(self.i2c_addr, bytearray([1 << SHIFT_BACKLIGHT]))

  def hal_backlight_off(self):
    """Allows the hal layer to turn the backlight off."""
    self.i2c.writeto(self.i2c_addr, bytearray([0]))
```

## >>> lcd_api.py [DARY21]

```
"""Provides an API for talking to HD44780 compatible character LCDs."""

import time

class LcdApi:
    """Implements the API for talking with HD44780 compatible character LCDs.
    This class only knows what commands to send to the LCD, and not how to get
    them to the LCD.

    It is expected that a derived class will implement the hal_xxx functions.
    """

    # The following constant names were lifted from the avrlib lcd.h
    # header file, however, I changed the definitions from bit numbers
    # to bit masks.
    #
    # HD44780 LCD controller command set

    LCD_CLR = 0x01              # DB0: clear display
    LCD_HOME = 0x02             # DB1: return to home position

    LCD_ENTRY_MODE = 0x04       # DB2: set entry mode
    LCD_ENTRY_INC = 0x02        # --DB1: increment
    LCD_ENTRY_SHIFT = 0x01      # --DB0: shift

    LCD_ON_CTRL = 0x08          # DB3: turn lcd/cursor on
    LCD_ON_DISPLAY = 0x04       # --DB2: turn display on
    LCD_ON_CURSOR = 0x02        # --DB1: turn cursor on
    LCD_ON_BLINK = 0x01         # --DB0: blinking cursor

    LCD_MOVE = 0x10             # DB4: move cursor/display
    LCD_MOVE_DISP = 0x08        # --DB3: move display (0-> move cursor)
    LCD_MOVE_RIGHT = 0x04       # --DB2: move right (0-> left)

    LCD_FUNCTION = 0x20         # DB5: function set
    LCD_FUNCTION_8BIT = 0x10    # --DB4: set 8BIT mode (0->4BIT mode)
    LCD_FUNCTION_2LINES = 0x08  # --DB3: two lines (0->one line)
    LCD_FUNCTION_10DOTS = 0x04  # --DB2: 5x10 font (0->5x7 font)
    LCD_FUNCTION_RESET = 0x30   # See "Initializing by Instruction" section

    LCD_CGRAM = 0x40            # DB6: set CG RAM address
    LCD_DDRAM = 0x80            # DB7: set DD RAM address

    LCD_RS_CMD = 0
    LCD_RS_DATA = 1

    LCD_RW_WRITE = 0
    LCD_RW_READ = 1

    def __init__(self, num_lines, num_columns):
        self.num_lines = num_lines
        if self.num_lines > 4:
            self.num_lines = 4
        self.num_columns = num_columns
        if self.num_columns > 40:
            self.num_columns = 40
        self.cursor_x = 0
        self.cursor_y = 0
        self.implied_newline = False
```

```
        self.backlight = True
        self.display_off()
        self.backlight_on()
        self.clear()
        self.hal_write_command(self.LCD_ENTRY_MODE | self.LCD_ENTRY_INC)
        self.hide_cursor()
        self.display_on()

    def clear(self):
        """Clears the LCD display and moves the cursor to the top left
        corner.
        """
        self.hal_write_command(self.LCD_CLR)
        self.hal_write_command(self.LCD_HOME)
        self.cursor_x = 0
        self.cursor_y = 0

    def show_cursor(self):
        """Causes the cursor to be made visible."""
        self.hal_write_command(self.LCD_ON_CTRL | self.LCD_ON_DISPLAY |
                               self.LCD_ON_CURSOR)

    def hide_cursor(self):
        """Causes the cursor to be hidden."""
        self.hal_write_command(self.LCD_ON_CTRL | self.LCD_ON_DISPLAY)

    def blink_cursor_on(self):
        """Turns on the cursor, and makes it blink."""
        self.hal_write_command(self.LCD_ON_CTRL | self.LCD_ON_DISPLAY |
                               self.LCD_ON_CURSOR | self.LCD_ON_BLINK)

    def blink_cursor_off(self):
        """Turns on the cursor, and makes it no blink (i.e. be solid)."""
        self.hal_write_command(self.LCD_ON_CTRL | self.LCD_ON_DISPLAY |
                               self.LCD_ON_CURSOR)

    def display_on(self):
        """Turns on (i.e. unblanks) the LCD."""
        self.hal_write_command(self.LCD_ON_CTRL | self.LCD_ON_DISPLAY)

    def display_off(self):
        """Turns off (i.e. blanks) the LCD."""
        self.hal_write_command(self.LCD_ON_CTRL)

    def backlight_on(self):
        """Turns the backlight on.

        This isn't really an LCD command, but some modules have backlight
        controls, so this allows the hal to pass through the command.
        """
        self.backlight = True
        self.hal_backlight_on()

    def backlight_off(self):
        """Turns the backlight off.

        This isn't really an LCD command, but some modules have backlight
        controls, so this allows the hal to pass through the command.
        """
        self.backlight = False
        self.hal_backlight_off()

    def move_to(self, cursor_x, cursor_y):
```

```
    """Moves the cursor position to the indicated position. The cursor
    position is zero based (i.e. cursor_x == 0 indicates first column).
    """
    self.cursor_x = cursor_x
    self.cursor_y = cursor_y
    addr = cursor_x & 0x3f
    if cursor_y & 1:
        addr += 0x40    # Lines 1 & 3 add 0x40
    if cursor_y & 2:    # Lines 2 & 3 add number of columns
        addr += self.num_columns
    self.hal_write_command(self.LCD_DDRAM | addr)

def putchar(self, char):
    """Writes the indicated character to the LCD at the current cursor
    position, and advances the cursor by one position.
    """
    if char == '\n':
        if self.implied_newline:
            # self.implied_newline means we advanced due to a wraparound,
            # so if we get a newline right after that we ignore it.
            self.implied_newline = False
        else:
            self.cursor_x = self.num_columns
    else:
        self.hal_write_data(ord(char))
        self.cursor_x += 1
    if self.cursor_x >= self.num_columns:
        self.cursor_x = 0
        self.cursor_y += 1
        self.implied_newline = (char != '\n')
    if self.cursor_y >= self.num_lines:
        self.cursor_y = 0
    self.move_to(self.cursor_x, self.cursor_y)

def putstr(self, string):
    """Write the indicated string to the LCD at the current cursor
    position and advances the cursor position appropriately.
    """
    for char in string:
        self.putchar(char)

def custom_char(self, location, charmap):
    """Write a character to one of the 8 CGRAM locations, available
    as chr(0) through chr(7).
    """
    location &= 0x7
    self.hal_write_command(self.LCD_CGRAM | (location << 3))
    self.hal_sleep_us(40)
    for i in range(8):
        self.hal_write_data(charmap[i])
        self.hal_sleep_us(40)
    self.move_to(self.cursor_x, self.cursor_y)

def hal_backlight_on(self):
    """Allows the hal layer to turn the backlight on.

    If desired, a derived HAL class will implement this function.
    """
    pass

def hal_backlight_off(self):
    """Allows the hal layer to turn the backlight off.
```

```
        If desired, a derived HAL class will implement this function.
        """
        pass

    def hal_write_command(self, cmd):
        """Write a command to the LCD.

        It is expected that a derived HAL class will implement this
        function.
        """
        raise NotImplementedError

    def hal_write_data(self, data):
        """Write data to the LCD.

        It is expected that a derived HAL class will implement this
        function.
        """
        raise NotImplementedError

    # This is a default implementation of hal_sleep_us which is suitable
    # for most micropython implementations. For platforms which don't
    # support `time.sleep_us()` they should provide their own implementation
    # of hal_sleep_us in their hal layer and it will be used instead.
    def hal_sleep_us(self, usecs):
        """Sleep for some time (given in microseconds)."""
        time.sleep_us(usecs)  # NOTE this is not part of Standard Python library, specific hal layers will need to override
this
```

## >>> funciones.py

```
from machine import Pin, ADC
import time
from time import ticks_ms

sensor0 = ADC(26) #ref del 0
sensor1 = ADC(27) #operacional derecho
sensor2 = ADC(28) #operacional izquierdo


def mide(sens):
    i=0
    x=[]
    a=0
    if sens == 0:
        for i in range(500):
            a =  sensor0.read_u16()
            x.append(a)
            time.sleep(2/1000)
        return x
    if sens == 1:
        for i in range(500):
            a =  sensor1.read_u16()
            x.append(a)
            time.sleep(2/1000)
        return x
    if sens == 2:
        for i in range(500):
            a =  sensor2.read_u16()
            x.append(a)
            time.sleep(2/1000)
        return x

def media(vec, zero):
    m=0
    for i in range(len(vec)):
        if(vec[i] >= 65500): #dejamos un margen con el 0 que tenemos de referencia, mejor que no se acerque
            return "SAT"
        elif(vec[i] <= zero):
            return "NEG"
        else:
            m=m+vec[i]
    m=m/len(vec)
    return m

def MAX(vec):
    m=0
    for i in range(len(vec)):
        if(vec[i] > m): #dejamos un margen con el 0 que tenemos de referencia, mejor que no se acerque
            m=vec[i]
    return m

def MIN(vec):
    m=65535
    for i in range(len(vec)):
        if(vec[i] < m): #dejamos un margen con el 0 que tenemos de referencia, mejor que no se acerque
            m=vec[i]
    return m

def refZero():
    zero = mide(0)
```

```
zero = media(zero,0)
return zero
```

# BIBLIOGRAFÍA


[DAMI23]: Firmware para microcontrolador Raspberry Pi Pico.
https://micropython.org/download/rp2-pico-w/

[HYLA20]: dump_mem.py, conversor HEX/ASCII.
https://github.com/dhylands/bioloid3/blob/master/bioloid/dump_mem.py

[BINA23]: Conversor ASCII/HEX online.
https://www.binaryhexconverter.com/hex-to-ascii-text-converter

[TEXA92]: Data Sheet TL082.
https://html.alldatasheet.com/html-pdf/28774/TI/TL081/67/3/TL081.html

[RASP23]: Data Sheet Raspberry Pi Pico.
https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf?_gl=1*c9z6h1*_ga*MTQxNTQzNTA3LjE2NzUxOTc2ODU.*_ga_22FD70LWDS*MTY4Njc1ODY5Mi4xLjAuMTY4Njc1ODY5Mi4wLjAuMA

[HYLA21]: esp8266_i2c_lcd.py
https://github.com/dhylands/python_lcd/blob/master/lcd/esp8266_i2c_lcd.py

[DARY21]: lcd_api.py
https://github.com/dhylands/python_lcd/blob/master/lcd/lcd_api.py