



**UNIVERSIDAD
DE GRANADA**

**TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA**

Validación de patrones generados por un producto electrónico microcontrolado

Autor

Jose Manuel García Cazorla

Directores

Andrés María Roldán Aranda



**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN**

—
Granada, Septiembre de 2025



Validación de patrones generados por un producto electrónico microcontrolado

Autor

Jose Manuel García Cazorla

Directores

Andrés María Roldán Aranda

Validación de patrones generados por un producto electrónico microcontrolado

Jose Manuel García Cazorla

Palabras clave: firmware, microcontrolador, Arduino, PWM, aplicación, front-end, C, Python, QT

Resumen

La industria del automóvil es un sector en constante desarrollo, siempre a la vanguardia de los avances tecnológicos para producir vehículos cada vez más sofisticados. Tratándose del medio de transporte más usado en la actualidad, cada pieza es sumamente importante: por muy prescindible que parezca, puede llegar a salvar vidas.

Uno de los componentes que más se pueden dar por sentado son los faros. Aunque se usen sólo una fracción del tiempo total que pasamos al volante, sin ellos no podríamos conducir de noche, o en situaciones en las que la visibilidad no sea óptima.

En este trabajo se presenta el análisis, diseño y desarrollo del firmware de PWM Box, un dispositivo que genera señales PWM configurables por el usuario con el objetivo de probar faros de vehículos. Asimismo, se implementa una interfaz gráfica que permite gestionar distintos perfiles y parámetros de configuración del dispositivo desde un ordenador.

Validation of patterns generated by an electronic microcontrolled product

Validación de patrones generados por un producto electrónico
microcontrolado

Jose Manuel García Cazorla

Keywords: firmware, microcontroller, Arduino, PWM, app, frontend, C, Python, QT

Abstract

The car industry is an ever-developing sector, always on the cutting edge of technological advances to produce increasingly sophisticated vehicles. Currently being the most used means of transportation, each of its components is extremely important: no matter how expendable it seems, it may even help to save lives.

One of the most easily overlooked components are headlights. Even though we only use them for a fraction of the time we spend on our cars, we wouldn't be able to drive at night or in low-visibility situations without them.

This project showcases the analysis, design and development of the firmware for the PWM Box, a device that generates user-configurable PWM signals intended for testing vehicle headlights. Additionally, a graphic interface is implemented, which allows to manage the device's profiles and other configuration parameters from a computer.

Yo, **Jose Manuel García Cazorla**, alumno del Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 15434710P, autorizo la ubicación de la siguiente copia de mi Trabajo de Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo.: Jose Manuel García Cazorla

Granada, a 5 de Septiembre de 2025.

D. Andrés María Roldán Aranda, Profesor del Departamento de Electrónica y Tecnología de Computadores de la Universidad de Granada.

Informan: Que el presente trabajo, titulado *Aplicación de control para testeador de faros de vehículos*, ha sido realizado bajo su supervisión por **Jose Manuel García Cazorla**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 5 de Septiembre de 2025.

El director:

Andrés María Roldán Aranda

Agradecimientos

En primer lugar, gracias a Andrés, mi tutor en este Trabajo de Fin de Grado, por guiarme en este proyecto ante todo inconveniente.

Gracias también a mis amigos, por hacer este proceso mucho más llevadero sin saberlo.

Y por último, lo más importante. Gracias a mis padres y a mi pareja, por entender siempre cómo me siento, por creer en mí, y por su incondicional apoyo a lo largo de toda esta etapa (y de mi vida). Sin ellos, este logro habría sido inalcanzable.

Índice general

1. Introducción	1
1.1. Contexto	1
1.2. Motivación	2
1.3. Objetivos	2
1.4. Estructura del documento	3
2. Estado del arte	5
2.1. Ingeniería inversa	5
2.1.1. Archivo principal	7
2.1.2. EEPROM	8
2.1.3. Entrada/Salida	9
2.1.4. LCD	9
2.1.5. Vista general	10
2.2. Tecnologías usadas	11
2.2.1. Sistema operativo	11
2.2.2. Lenguajes de programación	11
2.2.3. Entorno de desarrollo	13
3. Planificación temporal	15
3.1. Estimación	15
3.2. Tiempo real	16
4. Firmware	19
4.1. Análisis	19
4.2. Diseño	22
4.2.1. Menús	23
4.2.2. Entrada/Salida	23
4.2.3. Memoria EEPROM	24
4.2.4. General	27
4.3. Desarrollo	31
4.3.1. Menú de señales lentas	31
4.3.2. <i>Rotary encoder</i>	31
4.3.3. Serial control	32

4.3.4. Memoria EEPROM	33
4.4. Pruebas	34
5. Interfaz gráfica	37
5.1. Análisis	37
5.2. Diseño	41
5.2.1. PWM Types	45
5.2.2. JSON Manager	45
5.2.3. PWM Box	45
5.2.4. Main	45
5.2.5. General	47
5.3. Desarrollo	51
5.4. Pruebas	51
6. Presupuesto	53
7. Conclusiones y trabajos futuros	55
7.1. Conclusiones	55
7.2. Trabajos futuros	56

Índice de figuras

2.1. Dispositivo PWM Box desde arriba.	5
2.2. Lateral del dispositivo PWM Box.	6
2.3. Contenido de <i>PWM_BOX.RAR</i>	6
3.1. Diagrama de Gantt que muestra la organización de las distintas fases del proyecto a lo largo de las semanas.	16
4.1. Representación del funcionamiento del vector auxiliar al realizar distintas operaciones sobre la EEPROM.	26
4.2. Distribución del código en distintos archivos y módulos.	27
4.3. Ejecución del bucle principal.	29
4.4. Ejecución de las distintas interrupciones.	30
4.5. Contenido de la memoria EEPROM.	33
4.6. Señales PWM con distintas configuraciones.	35
4.7. Una de las secuencias de señales lentas pre-configuradas.	35
4.8. Otra de las secuencias de señales lentas pre-configuradas.	35
5.1. Primera iteración del diseño de la interfaz.	42
5.2. Otra iteración del diseño de la interfaz.	42
5.3. Otra iteración del diseño de la interfaz.	43
5.4. Diseño final de la interfaz.	43
5.5. A la izquierda, ventana para exportar ranuras. A la derecha, ventana para enviar ranuras al dispositivo	44
5.6. Flujo de ejecución al recargar la conexión con el dispositivo.	47
5.7. Flujo de ejecución al enviar perfiles al dispositivo.	48
5.8. Flujo de ejecución al exportar perfiles al equipo.	48
5.9. Flujo de ejecución al importar perfiles desde el equipo.	49
5.10. Flujo de ejecución al mostrar y modificar la contraseña.	49
5.11. Flujo de ejecución al establecer el perfil por defecto.	50

Índice de cuadros

4.1. RF-1. Guardar perfiles	20
4.2. RF-2. Cargar perfiles	20
4.3. RF-3. Eliminar perfiles	20
4.4. RF-4. Guardar la contraseña	21
4.5. RF-5. Guardar el brillo de la pantalla.	21
4.6. RF-6. Mostrar secuencias disponibles.	21
4.7. RF-7. Ejecutar secuencias lentas.	21
4.8. RF-8. Parar la ejecución de secuencias lentas.	22
4.9. RNF-1. Manejabilidad.	22
4.10. RNF-2. Rendimiento.	22
4.11. RNF-3. Extensibilidad.	22
5.1. RF-1. Conectar con el dispositivo.	37
5.2. RF-2. Obtener la información básica del dispositivo.	38
5.3. RF-3. Obtener la configuración general del dispositivo.	38
5.4. RF-4. Obtener las ranuras guardadas en el dispositivo.	38
5.5. RF-5. Enviar configuración general al dispositivo.	38
5.6. RF-6. Enviar ranuras al dispositivo.	38
5.7. RF-7. Mostrar la información básica del dispositivo.	39
5.8. RF-8. Mostrar la configuración general del dispositivo.	39
5.9. RF-9. Mostrar los parámetros de las señales generadas.	39
5.10. RF-10. Crear ranuras nuevas.	39
5.11. RF-11. Exportar ranuras al equipo.	40
5.12. RF-12. Importar ranuras del equipo.	40
5.13. RF-13. Visualizar la configuración de las ranuras.	40
5.14. RF-14. Modificar los parámetros de las ranuras cargadas.	40
5.15. RNF-1. Manejabilidad.	40
5.16. RNF-2. Rendimiento.	41
5.17. RNF-3. Aspecto atractivo.	41

Todo el código fuente desarrollado para este Trabajo de Fin de Grado, incluyendo el de esta memoria, se encuentra disponible en el siguiente repositorio de GitHub:

<https://github.com/xemanue/TFG>



Capítulo 1

Introducción

Como es costumbre, esta memoria merece ser empezada por el principio: el “qué” y el “por qué” de este proyecto. Esa es la función que cumplirá esta introducción, en la que se contextualizarán los aspectos fundamentales del trabajo, así como la motivación detrás de su realización. A partir de ellos, se definirán los objetivos específicos que se tratarán de cumplir, y por último se describirá brevemente la estructura de este documento.

1.1. Contexto

Un microcontrolador es un circuito integrado programable que está compuesto de distintos bloques funcionales que cumplen tareas específicas. Este incluye en su interior los tres componentes esenciales de un ordenador (una CPU, memoria y mecanismos de entrada y salida), convirtiéndolo funcionalmente en un ordenador a pequeña escala.

A pesar de su tamaño, son muy potentes, consumiendo menos energía que un ordenador convencional y a un precio mucho más bajo. Esto permite integrarlos en todo tipo de dispositivos, desde electrodomésticos o teléfonos móviles hasta aquellos usados en la automatización industrial y en la industria aeroespacial. Hoy en día son esenciales para muchas de las tareas que realizamos de manera cotidiana.

Sin embargo, a pesar de sus numerosos beneficios, sus recursos son limitados, lo cual los hace menos apropiados para aplicaciones que necesiten una mayor cantidad de memoria o de procesamiento. Es por ello que usarlos de forma conjunta con un ordenador tradicional puede ser un factor clave en algunos casos.

1.2. Motivación

Este proyecto se realiza a partir de la necesidad de Valeo, una empresa distribuidora en el ámbito automovilístico, de proporcionar a sus fabricantes una forma de probar el funcionamiento de los faros de distintos vehículos.

El laboratorio de GranaSAT, del que el tutor de este trabajo forma parte, fue el encargado de proporcionar una solución a dicha urgencia: usar un dispositivo basado en un microcontrolador para producir las señales PWM que alimentan dichos faros.

Esta ha sido presentada como Trabajo de Fin de Grado en años anteriores. Primero por Luis Sánchez, autor de la PCB y de todo el apartado electrónico, y luego por Rubén Sánchez, que se encargó de desarrollar el firmware del dispositivo.

Este dispositivo, al que llamamos PWM Box, ha sido usado por Valeo de forma satisfactoria. Sin embargo, se han vuelto a poner en contacto con GranaSAT para expresar una nueva necesidad: la capacidad de almacenar distintos perfiles de configuración para el dispositivo.

Para ello, se plantea en este trabajo la implementación de nuevas funcionalidades en el dispositivo que permitan resolver esta cuestión. Asimismo, se pretende proporcionar a la empresa una interfaz gráfica que les permita gestionar estas configuraciones sin tener que depender exclusivamente de la limitada memoria del microcontrolador.

A su vez, a nivel personal, esta tarea es idónea para complementar las competencias que he adquirido a lo largo de la mención de Ingeniería de Computadores del Grado. Por un lado, me permite poner en práctica lo aprendido en un proyecto real, mientras que por otro, constituye una forma de acercarme a un desarrollo algo más a alto nivel.

1.3. Objetivos

A continuación se definen los requisitos generales y específicos de este TFG, obtenidos a partir del contexto proporcionado en esta introducción.

- **OG-1.** Mejorar el actual firmware del PWM Box.
 - **OE-1.1.** Realización de una ingeniería inversa de la versión actual del firmware, con el objetivo de localizar y corregir posibles errores existentes.
 - **OE-1.2.** Desarrollar una funcionalidad que permita la utilización de la memoria EEPROM del microcontrolador para almacenar perfiles de configuración.

- **OE-1.3.** Incluir un nuevo modo de funcionamiento que permita probar faros que requieran señales de una frecuencia inferior a la que el dispositivo es capaz de generar.
- **OG-2.** Implementación de una interfaz gráfica complementaria.
 - **OE-2.1.** Integrar la aplicación con el dispositivo, permitiendo el intercambio de información entre ambos.
 - **OE-2.2.** Implementar los mecanismos necesarios para permitir el envío de perfiles de configuración desde la interfaz al PWM Box.
 - **OE-2.3.** Incluir en la interfaz elementos que permitan visualizar y modificar las distintas configuraciones que esta almacene.
 - **OE-2.4.** Permitir la exportación e importación de perfiles de la interfaz al disco del ordenador.

Se determinan, adicionalmente, los siguientes objetivos de aprendizaje:

- **OA-1.** Poner en práctica en un proyecto real los conocimientos adquiridos durante el Grado acerca de la programación a bajo nivel.
- **OA-2.** Aprender sobre el diseño y desarrollo de una aplicación de escritorio.
- **OA-3.** Adquirir destreza en algún lenguaje de programación que no haya usado durante la carrera.
- **OA-4.** Llevar a cabo la integración de distintos sistemas independientes en un mismo producto final.

1.4. Estructura del documento

Este documento se divide en distintos apartados, centrándose cada uno en un aspecto distinto del tema que trata:

- **Introducción:** El primer apartado, en el cual nos encontramos, sirve de introducción al proyecto. Proporciona información general del desarrollo que se va a abordar, proporcionando contexto y definiendo a grandes rasgos las tareas que se esperan cumplir.
- **Estado del arte:** En segundo lugar, se realizará un análisis del estado del proyecto antes de empezar a trabajar en él. Se tratará de describir de forma detallada las características de las versiones anteriores del firmware, identificando áreas en las que mejorar y comenzando a tomar algunas decisiones de cara a la propuesta.

- **Planificación temporal:** En este apartado se describirá la metodología seguida para llevar a cabo el trabajo, planificando de antemano cuánto tiempo invertir en cada una de sus partes.
- **Firmware:** Constituye el principio de la propuesta. En él se desarrollan las distintas fases en las que se ha decidido enfrentar esta parte del proyecto.
- **Interfaz:** Segunda parte de la propuesta, en la que se abordan los distintos aspectos definidos para la implementación de la interfaz gráfica.
- **Presupuesto:** En este apartado se realiza una breve estimación del coste total del desarrollo del proyecto, desglosado según su procedencia.
- **Conclusiones y trabajo futuro:** Finalmente, se concluirá la memoria reflexionando sobre el cumplimiento de los objetivos fijados. También se mencionarán algunos aspectos en los que se considera que hay cabida para mejoras, en caso de que continúe trabajando en el proyecto.

Capítulo 2

Estado del arte

Como se ha explicado en la [introducción](#), este trabajo continúa el comenzado por el alumno Rubén Sanchez en cursos anteriores. Por ello, el primer paso a realizar como revisión del estado del arte es una ingeniería inversa de su desarrollo, con el fin de identificar aspectos en los que se puede mejorar.

2.1. Ingeniería inversa

Al comenzar el trabajo, se recibe por parte del tutor el prototipo del dispositivo, que se puede ver en las figuras [2.1](#) y [2.2](#).



Figura 2.1: Dispositivo PWM Box desde arriba.

Este está compuesto por un Arduino Mega 2560, una pantalla LCD de



Figura 2.2: Lateral del dispositivo PWM Box.

4 líneas, un *rotary encoder* y 8 conectores de salida. Por cada uno de estos 8 conectores se produce una salida PWM cuyos parámetros (encendido, frecuencia, ciclo de trabajo y fase) pueden ser configurados por el usuario, pensada para conectar y probar los distintos modos del faro del vehículo.

Por otro lado, se recibe un archivo comprimido que contiene el código existente del proyecto, así como alguna documentación escrita por sus previos contribuyentes. Estos archivos se muestran en la figura 2.3.

```
▶ ls
avrduude.conf          PWM_BOX.ino
backup_eeprom.hex       PWM_BOX.ino.hex
config.h                PWM_BOX.ino.with_bootloader.hex
EEPROM_8channels0_1_2_3.hex  PWM_BOX_V21_EEPROM_READ.bat
EEPROM_8channels0_1.hex   PWM_BOX_V21_EEPROM_WRITE.bat
EEPROM_8channels_m.hex   PWM_BOX_V21_programROM.bat
EEPROM_blancked.hex     PWM_BOX_V21_programROM+EEPROM_avrdragon.bat
EEPROM_control.c        PWM_BOX_V21_programROM+EEPROM.bat
EEPROM_control.h        pwm_gen.c
IO_control.c            pwm_gen.h
IO_control.h            README.md
LCD.c                  README.md.bak
LCD.h                  reverse assembler
LCD_menu.c              uart_control.c
LCD_menu.h              uart_control.h
LIFO_mem.c              virtual_PWM.c
LIFO_mem.h              virtual_PWM.h
```

Figura 2.3: Contenido de *PWM_BOX.RAR*.

Con esto, se procede a realizar una clasificación básica de los distintos ficheros a modo de primera toma de contacto con el proyecto. Según su contenido y utilidad, se pueden distinguir los siguientes tipos de archivos:

- **Archivos HEX:** Estos contienen datos en formato hexadecimal. Por

el nombre, se pueden distinguir entre los que contienen datos para la memoria no volátil del microcontrolador (EEPROM), y los que contienen el programa en sí.

- **Archivos BAT:** Se tratan de archivos de comandos para Windows. Examinando su contenido, se puede ver que todos ellos ejecutan comandos de un programa llamado AVRDUDE.
AVRDUDE [1] es una utilidad multiplataforma para la programación de microcontroladores AVR, entre los que se encuentran los usados por Arduino. Permite todo tipo de operaciones de carga y descarga de datos de las distintas memorias del microcontrolador. Es una herramienta usada muy comúnmente, que suele funcionar de *back-end* en otros entornos.
- **Archivos de código:** Por un lado tendríamos las cabeceras y su implementación, en los usuales archivos H y C. Por otro lado, encontramos un archivo INO, que es el tipo usado por Arduino IDE para distinguir que se trata del archivo principal del proyecto. Este contiene las funciones `setup()` y `loop()`, que definen el flujo básico del programa: la primera se ejecuta una vez al inicio, y la segunda conforma el bucle principal de ejecución.

Tras esta primera distinción, se pasa a hacer un análisis del código y de su funcionamiento, usando como apoyo la documentación de Rubén. Según la misma, se puede dividir el programa en los grupos que se describen a continuación.

2.1.1. Archivo principal

Consiste en el archivo principal INO que se mencionó anteriormente. En él, encontramos algunos elementos usados para determinar el flujo de ejecución del programa:

- En primer lugar tenemos una cola de eventos, que recoge las acciones del usuario y permite al programa procesarlas en orden. Se desarrollará en una sección posterior.
- Un contador de tiempo, que mide cuánto lleva el usuario sin interactuar con el dispositivo para mostrar u ocultar un salva-pantallas.
- El uso de distintas interrupciones para el manejo de la entrada del usuario, la generación de la señal PWM de salida y la comunicación por UART.

Una cosa a comentar sobre el actual flujo del programa es el uso de interrupciones. Su uso es muy necesario para la generación de las señales PWM. Una interrupción se genera a una cierta frecuencia, y, dependiendo de los parámetros que introduzca el usuario, las señales cambian de estado cuando les corresponda. Esto asegura que los cambios se realizan en el momento preciso, mientras que si se gestionaran en el bucle principal, el momento exacto de las actualizaciones sería más incierto.

Sin embargo, la alta frecuencia de las interrupciones podría provocar una “sobrecarga” (*interrupt overload*). Esta provocaría que otras tareas en ejecución, en este caso el bucle principal, no recibiera el suficiente tiempo de CPU, afectando al rendimiento. Esta condición puede también darse si, aun manteniendo un bajo número de interrupciones, las rutinas que estas ejecutan son demasiado largas.

Por todo ello, el uso de interrupciones para manejar la entrada de usuario a través del codificador rotatorio parece, tras un primer análisis, acertado. Su frecuencia será varias magnitudes menor que las destinadas a atender las señales de salida, de forma que no aportarán al problema anteriormente mencionado. Al mismo tiempo, se asegurarán de que ningún *input* del usuario se pierde debido a que la CPU se encuentre ocupada. [11]

Buffer de eventos

Se trata de una cola circular estándar, en la que se van incluyendo datos de tipo evento. Estos consisten de un entero que determina el tipo de evento, junto a dos parámetros opcionales. Estos últimos parecen no ser utilizados en el resto del programa. Se distinguen eventos para giros del *rotary encoder*, para su pulsación y para la llegada de datos a la UART.

El uso de un buffer de eventos aporta un mecanismo más para evitar la comentada sobrecarga por interrupciones. Con él, las funciones que las procesan pueden dedicarse únicamente a añadir el evento que corresponda a la cola, evitando rutinas demasiado largas. Será luego el bucle principal el que se encargue de ejecutar la acción vinculada al evento concreto, cuando la CPU pueda dedicarle tiempo.

2.1.2. EEPROM

Se trata de un archivo que sienta las bases para el guardado de datos en la memoria no volátil del microcontrolador. Este punto se distingue como uno de los que quedó fuera del desarrollo anterior, ya que únicamente aparecen implementadas algunas funciones básicas para leer posiciones determinadas de la memoria, que llaman directamente a otras de la librería de AVR.

De la breve implementación presente en este punto se puede comentar una cosa, y es que las lecturas y escrituras se están realizando proporcionando directamente la posición de memoria. Esto es un claro punto a mejorar, puesto que mantener estas posiciones guardadas puede ser tedioso, y disminuir la legibilidad del código. Una forma más apropiada de tratarlo sería definir la estructura de la memoria, de forma que los accesos a la misma puedan realizarse usando variables.

2.1.3. Entrada/Salida

En él se definen distintas funciones relacionadas con el codificador rotatorio y el manejo de eventos. También se distingue el prototipo de una función que serviría para decodificar un mensaje recibido por el puerto serie. Sin embargo, la lógica a bajo nivel del mismo no está implementada, por lo que actualmente no se le está dando uso.

De este módulo cabe destacar que el funcionamiento del *rotary encoder* no es del todo correcto, generando en ocasiones dobles pulsaciones y no detectando algunos de los giros.

2.1.4. LCD

Se trata, con diferencia, del módulo más extenso de todo el programa. En él aparecen definidas distintas funciones que determinan el contenido de la pantalla según las acciones que vaya realizando el usuario. Podemos distinguir los siguientes menús:

- **Lista:** Es el menú principal del sistema. En él se muestran las diferentes señales a configurar, así como las opciones de guardar y cargar configuraciones en la EEPROM (no operativas aún).
- **PWM:** Permite la configuración de los 4 parámetros principales del PWM que se seleccione en la pantalla anterior.
- **Contraseña:** Permite bloquear algunas características del dispositivo, como las que alteran el estado de la memoria EEPROM, tras una contraseña, evitando el acceso de usuarios no autorizados.

Para cada uno de los menús mencionados, se encuentran en este archivo las definiciones de cuatro funciones, que permiten realizar operaciones básicas sobre el menú al que pertenecen. Se tratan de las siguientes:

- **Actualización de la pantalla:** Encontramos una para cada menú del firmware. Se encarga de dibujar el contenido del LCD dependiendo

del contexto, como el lugar en el que se encuentre el cursor, la opción que se encuentre activa, etc.

- **Pulsación del botón:** Determina la acción a realizar cuando el usuario accione el *rotary encoder*, basándose una vez más en la posición del cursor.
- **Desplazamiento hacia arriba:** Concreta el cambio a realizar en las variables internas del menú cuando el usuario gire el control hacia la izquierda.
- **Desplazamiento hacia abajo:** Realiza la misma función que el punto anterior, pero para un giro en la dirección opuesta.

De esta parte se extraen varios puntos a mejorar. Por un lado, el archivo es demasiado extenso como para ser navegado cómodamente. Contiene la implementación de todas las funciones de todos los menús del sistema sin demasiada organización, incluso aunque durante la ejecución sólo se usen en cada momento las del menú que se encuentre activo. De la misma forma, como se ha señalado de manera consciente, encontramos código redundante dentro de las funciones de un mismo menú, como pone de manifiesto manejar las dos direcciones de giro del *rotary encoder* en funciones separadas.

Por otro lado, las variables que hacen referencia a los distintos menús se encuentran agrupadas en estructuras independientes. De esta forma se consigue que no haya conflicto entre ellas, pero hacen el código mucho menos legible debido a la longitud de cada una de sus referencias.

Esto trae consigo numerosos problemas que dificultan el mantenimiento del código, tanto a la hora de corregir errores existentes como al tratar de extender las funcionalidades del dispositivo, que son dos de los **objetivos** de este trabajo.

Adicionalmente, se ha detectado el incorrecto funcionamiento del *rotary encoder* en determinados menús, en los que las direcciones de giro están invertidas.

2.1.5. Vista general

Esta versión del firmware establece una buena base para el desarrollo. Define un flujo de ejecución sólido, y nos permite centrar los esfuerzos en realizar las correcciones y mejoras oportunas sobre los distintos módulos de manera individual. De esta forma, se divide el trabajo en objetivos concretos que evitan grandes modificaciones que no sean estrictamente necesarias, así como consecuentes inversiones innecesarias de tiempo.

A su vez, de manera global, se han detectado en el código algunos puntos a mejorar en cuanto a legibilidad, presentación y manejabilidad del código. Sin embargo, este es un aspecto subjetivo, por lo que no se cubrirá en esta memoria.

2.2. Tecnologías usadas

En esta sección destacaré las distintas herramientas y tecnologías involucradas en el desarrollo del proyecto, aportando algunas posibles alternativas y destacando la opción elegida en cada caso.

2.2.1. Sistema operativo

Hasta ahora, todo el código usado en el proyecto se ha desarrollado en Windows. Dada la necesidad de comprobar el funcionamiento del producto final tanto en este como en Linux (establecida como objetivo en la sección 1.3), el sistema operativo a usar no es un aspecto decisivo a tener en cuenta en este caso.

Sin embargo, para trabajar más cómodamente se ha decidido usar Linux por preferencia personal. Esto puede resultar un inconveniente a la hora de usar los archivos BAT proporcionados al comienzo del proyecto. Sin embargo, como se comentó en la sección anterior, el programa al que estos hacen referencia es multiplataforma, por lo que bastaría con transferirlos a un *script* en Bash si fuese necesario.

En general, al no ser algo relevante en el proyecto, consideramos que la familiaridad con el entorno de desarrollo resulta una prioridad, debido a la capacidad que proporciona para solucionar cualquier imprevisto que pueda surgir en el curso del trabajo. A la hora de probar el código, se puede recurrir a una máquina virtual, o a una partición con otro sistema si se prefiere.

2.2.2. Lenguajes de programación

Firmware

El lenguaje de programación usado actualmente en el proyecto es C. C es un lenguaje de programación de propósito general diseñado originalmente para su uso en el sistema operativo UNIX, estando este, junto a la mayoría de programas usados en él, escritos en C. Sin embargo, su uso a lo largo del tiempo ha transcendido su objetivo inicial, convirtiéndose en un lenguaje muy ampliamente usado en la actualidad, debido entre otros factores a la eficiencia del código que produce y a su portabilidad. Dispone de estructuras

típicas de los lenguajes de alto nivel, permitiendo a su vez un gran control del sistema a bajo nivel. Esto lo convierte en un lenguaje muy versátil, pudiendo llegar a ser más conveniente y efectivo en la ejecución de diversas tareas que otras alternativas más potentes. [4]

Estando esta parte del proyecto basada en la programación de un Arduino, otra opción disponible sería usar C++. Este surgió con la intención de extender el lenguaje de programación C a través de la inclusión de características de la programación orientada a objetos. A lo largo del tiempo, se fueron también incorporando a él otras funciones propias de la programación genérica, de la programación estructurada, facilidades para la programación a bajo nivel... Es por ello que se suele considerar a C++ un lenguaje de programación multiparadigma. C++ es un lenguaje muy completo, manteniendo una eficiencia similar a la de C pero añadiendo un gran número de características de lenguajes de más alto nivel, siendo apropiado para el desarrollo de programas, videojuegos o servidores.

Teniendo en cuenta que el objetivo actual de este ámbito es añadir algunas características y pulir la implementación del firmware, se considera más apropiado continuar trabajando en el lenguaje con el que se empezó el proyecto, es decir, C.

Interfaz

Para el desarrollo de la interfaz se necesita una librería gráfica que sea compatible tanto con Windows como con Linux, dado uno de los **objetivos** definidos anteriormente. Debido a ello, así como a experiencias previas, se ha optado por el uso de Qt.

Qt se trata de un *framework* multiplataforma que permite la creación de interfaces gráficas compatibles con Linux, Windows, macOS y Android (entre otros) sin necesidad de modificar el código base. Qt proporciona licencias comerciales, pero también está disponible una versión de código abierto a través de *Qt Project*. Su uso está muy extendido en la comunidad de Linux, siendo uno de los ejemplos más representativos KDE Plasma [7], un entorno gráfico *open-source* incluido en la mayor parte de las distribuciones más usadas.

Qt fue creado para ser usado con C++, pero en la actualidad tiene soporte para otros lenguajes de programación, como Python. Esta versión resulta especialmente interesante para este proyecto, debido a que la gran extensibilidad de este lenguaje podría facilitar la integración de la aplicación con el dispositivo.

2.2.3. Entorno de desarrollo

Firmware

Para la programación del firmware, la opción más directa sería usar el IDE proporcionado por Arduino. Este incluye por defecto todas las opciones necesarias para trabajar con uno de sus microcontroladores, como la detección automática de dispositivos, la posibilidad de compilar y cargar el código pulsando un botón y un monitor del puerto serie, por mencionar algunas. Sin embargo, tras haberlo usado en alguna de las asignaturas del Grado, considero que le faltan algunas características deseables para un proyecto más complejo como este, como puede ser un explorador de archivos integrado, la posibilidad de incluir ficheros de distintos directorios, o la completación de código.

Es por ello que finalmente se ha decidido usar un editor más genérico, como Visual Studio Code. Este es un editor *open-source* muy ligero y personalizable para todo tipo de necesidades. Una parte de esta personalización la consigue gracias a un soporte nativo de extensiones, que permite a cualquier usuario que lo desee ampliar sus características y publicar su creación en un “mercado de extensiones” completamente gratuito. Un ejemplo de estas extensiones es *PlatformIO*.

PlatformIO es un entorno de programación integrado en VSCode, que proporciona numerosos instrumentos que facilitan la programación de sistemas con microcontroladores. Algunos de ellos son justamente los mencionados anteriormente: un monitor del puerto serie, mecanismos para compilar y cargar código con solo pulsar un botón, etc.

Otras extensiones que también se usarán a lo largo del desarrollo son la extensión oficial de GitHub, que permite sincronizar los cambios con un repositorio remoto desde la propia interfaz de VSCode, o la extensión de Doxygen, que facilita la generación de documentación a través de *snippets*.

Interfaz

Para el desarrollo de la aplicación, Qt permite el uso de archivos UI, que facilitan la declaración de los distintos elementos de la interfaz. Estos archivos usan formato XML, por lo que pueden ser algo difíciles de editar directamente. Por ello, se hará uso de la herramienta *Qt Designer*, en la que se pueden colocar y configurar los elementos de una forma más visual, facilitando enormemente el proceso.

En cuanto al desarrollo, VSCode proporciona también una extensión que añade soporte para Python, por lo que se plantea usar este mismo IDE. De esta forma, se facilita la realización de pruebas de integración, al usar un

mismo entorno.

Capítulo 3

Planificación temporal

3.1. Estimación

Se dispone de un tiempo limitado para la realización de este proyecto. Planificar correctamente en qué invertirlo antes de comenzar a trabajar es muy importante, ya que puede prevenir un malgasto innecesario del mismo en tareas que no aporten suficiente al avance del desarrollo, así como proporcionar guías de actuación ante cualquier imprevisto que pueda surgir durante el proceso.

El Trabajo de Fin de Grado proporciona al estudiante 12 créditos ECTS en el Grado. Según la normativa de la Universidad de Granada, un crédito ECTS equivale a entre 25 y 30 horas de trabajo, por lo que podemos estimar que se espera en el TFG una inversión de unas 300 horas como mínimo. El segundo cuatrimestre del curso académico 24/25 tiene una duración de 15 semanas, lo cual resultaría en unas 20 horas de trabajo por semana, o unas 4 horas al día.

Teniendo en cuenta los requisitos del proyecto, junto a este análisis realizado, se llega a la conclusión de que el modelo más compatible es un híbrido entre el modelo en cascada y la metodología ágil, siguiendo un desarrollo basado en funcionalidades.

En primer lugar, se planificará el trabajo como un proceso lineal, lo cual encaja con las limitaciones temporales, y permite dividir el proceso de desarrollo en dos grandes fases: una para el dispositivo, y otra para la interfaz, para no desaprovechar el tiempo desarrollando el aspecto visual de características que aun no existen. Después, se dividirá cada fase en distintas funcionalidades clave, que se irán implementando y probando una a una.

De esta forma, el avance en el proyecto es continuo, y se aprovechan las características ventajosas de ambos métodos. Por el lado del modelo

en cascada, se evita invertir demasiado tiempo en la planificación de los distintos *sprints*, de acuerdo al limitado tiempo disponible. En lo que a la metodología ágil respecta, se aprovecha la frecuente realización de pruebas, que asegura la calidad del producto final.

Representando las distintas fases del modelo en cascada, obtendríamos una planificación parecida a la siguiente:

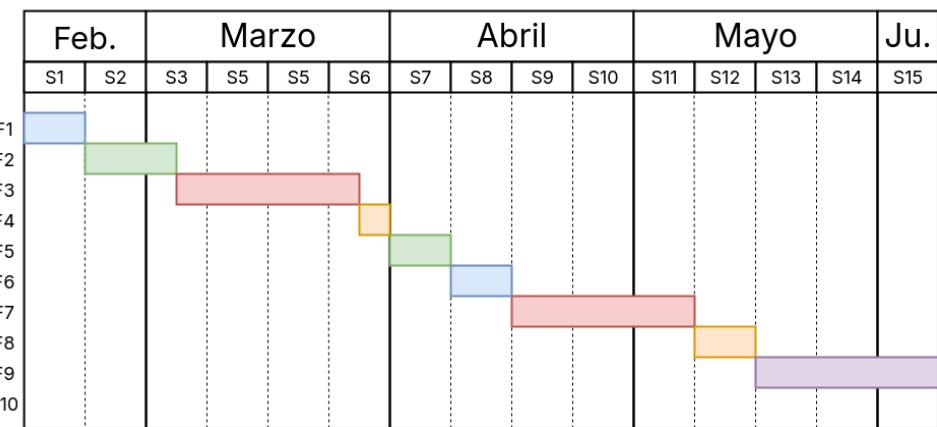


Figura 3.1: Diagrama de Gantt que muestra la organización de las distintas fases del proyecto a lo largo de las semanas.

- Las fases de color azul representan la definición de los requisitos de cada parte del trabajo.
- Las verdes corresponden al tiempo de diseño.
- Las secciones rojas son las de desarrollo.
- Las fases de pruebas vienen señaladas en color naranja, siendo la segunda un poco más extensa para poder hacer pruebas de integración entre los dos sistemas.
- En morado está marcada la fase de documentación, que incluye tanto la del código como la generación de esta memoria.

Las tareas a realizar en cada fase serán detalladas más a fondo en sus capítulos correspondientes.

3.2. Tiempo real

En la práctica, el cumplimiento de esta planificación no ha sido posible, debido a haber tenido que compaginar el desarrollo del proyecto con

unas prácticas extracurriculares, y un posterior puesto de trabajo a tiempo completo.

Esto, junto a un pobre manejo del tiempo en algunas fases, ha provocado un avance más lento de lo esperado. Dichas desalineaciones han causado, a su vez, una mayor inversión de tiempo de la esperada en el proyecto, ascendiendo aproximadamente a unas 450 horas a lo largo de 5 meses.

Capítulo 4

Firmware

4.1. Análisis

El primer paso para comenzar a trabajar en el firmware del dispositivo es realizar un análisis en el que se concreten los requisitos de las funcionalidades a incorporar al sistema.

Como se comentó en la sección 1.2, este proyecto es producto de las necesidades de Valeo, una empresa distribuidora de partes de vehículos. Junto a estas, pretenden también proporcionar a sus fabricantes herramientas para probar su correcto funcionamiento. Una de estas herramientas es el PWM Box.

En base a esto, se puede determinar que nuestro producto será usado principalmente en un entorno similar a una fábrica de coches o un taller mecánico. El perfil de su usuario principal será, por lo tanto, el de alguien que no necesariamente tenga conocimientos tecnológicos. Si se espera cierta familiaridad con dispositivos del entorno industrial, con controles similares al nuestro y una interfaz del mismo estilo.

Teniendo en cuenta las necesidades de la empresa y el perfil de este usuario, podemos definir los siguientes requisitos adicionales:

Gestión de perfiles

Cuadro 4.1: RF-1. Guardar perfiles.

ID	RF-1
Nombre	Guardar perfiles
Descripción	El sistema debe permitir al usuario guardar la configuración activa en forma de perfil, que debe mantenerse en la memoria tras reiniciar el dispositivo. Esta operación debe estar protegida con contraseña, de forma que solo los usuarios autorizados puedan realizarla.
Prioridad	Alta

Cuadro 4.2: RF-2. Cargar perfiles.

ID	RF-2
Nombre	Cargar perfiles
Descripción	El sistema debe permitir al usuario cargar los perfiles almacenados en la memoria, estableciéndolos como activos. Esta operación debe estar protegida con contraseña, de forma que solo los usuarios autorizados puedan realizarla.
Prioridad	Alta

Cuadro 4.3: RF-3. Eliminar perfiles.

ID	RF-3
Nombre	Eliminar perfiles
Descripción	El sistema debe permitir al usuario eliminar perfiles que estén actualmente almacenados en la memoria. Esta operación debe estar protegida con contraseña, de forma que solo los usuarios autorizados puedan realizarla.
Prioridad	Alta

Gestión de la configuración del dispositivo

Cuadro 4.4: RF-4. Guardar la contraseña.

ID	RF-4
Nombre	Guardar la contraseña
Descripción	El sistema debe almacenar la contraseña establecida en la memoria, de forma que esta se mantenga entre reinicios.
Prioridad	Alta

Cuadro 4.5: RF-5. Guardar el brillo de la pantalla.

ID	RF-5
Nombre	Guardar el brillo de la pantalla
Descripción	Para la comodidad del usuario, el sistema debe almacenar la configuración de brillo del LCD en la memoria, de forma que esta se mantenga entre reinicios.
Prioridad	Media

Funcionalidad de señales lentas

Cuadro 4.6: RF-6. Mostrar secuencias disponibles.

ID	RF-6
Nombre	Mostrar secuencias disponibles
Descripción	El sistema debe permitir al usuario consultar las secuencias lentas disponibles para ejecutar. Este menú debe ser independiente al resto del sistema.
Prioridad	Alta

Cuadro 4.7: RF-7. Ejecutar secuencias lentas.

ID	RF-7
Nombre	Ejecutar secuencias lentas
Descripción	El sistema debe permitir al usuario ejecutar las distintas secuencias lentas proporcionadas.
Prioridad	Alta

Cuadro 4.8: RF-8. Parar la ejecución de secuencias lentas.

ID	RF-8
Nombre	Parar la ejecución de secuencias lentas
Descripción	El sistema debe permitir al usuario parar la ejecución de la secuencia activa en cualquier momento.
Prioridad	Alta

Requisitos no funcionales

Cuadro 4.9: RNF-1. Manejabilidad.

ID	RNF-1
Nombre	Manejabilidad
Descripción	El sistema ha de ser sencillo e intuitivo para los usuarios.
Prioridad	Alta

Cuadro 4.10: RNF-2. Rendimiento.

ID	RNF-2
Nombre	Rendimiento
Descripción	El sistema debe ser fluido y responder a las acciones del usuario sin demoras.
Prioridad	Alta

Cuadro 4.11: RNF-3. Extensibilidad.

ID	RNF-3
Nombre	Extensibilidad
Descripción	El sistema debe estar diseñado de forma que sea sencillo añadir nuevas características.
Prioridad	Media

4.2. Diseño

Al tratarse de un desarrollo en proceso, el diseño de la arquitectura general del sistema ya estaba realizado. En este aspecto, los cambios a realizar son los derivados de la redistribución del código de algunas partes, que como se determinó en la sección 2.1, dificultaba la incorporación de nuevas características.

4.2.1. Menús

El principal aspecto a rediseñar está en la parte del sistema que muestra los distintos menús al usuario. La idea es separar la actual implementación de los distintos menús en archivos independientes.

Por un lado, se creará un archivo de control, que será al que se haga referencia desde el resto del programa para hacer operaciones como cambiar de menú, actualizar la pantalla, cambiar el brillo del LCD, etc. Cuando se llamen a funciones que dependan de la pantalla que esté activa en ese momento, él será el que se encargue de delegar la tarea al menú que corresponda.

Por otro lado, cada menú disponible tendrá un archivo separado en el que se implementarán como mínimo las versiones correspondientes de las 3 operaciones básicas (actualizar pantalla, girar *rotary encoder* y pulsar *rotary encoder*). Esto permite que, en menús que requieran funciones específicas (como será el caso del menú de señales lentas que se pretende añadir), estas funciones sólo sean accesibles desde el menú que corresponda.

En la figura 4.2 incluida más adelante podemos ver el resultado de esta división. Se destaca también la inclusión de un nuevo tipo de menú, el menú del sistema. Este se encargará de mostrar la secuencia de inicio y el salvapantallas, tareas que estaban antes también unificadas con el resto. También será útil una vez se incluya el almacenamiento de perfiles en la memoria, pues permitirá mostrar un aviso cuando se llene la memoria.

4.2.2. Entrada/Salida

Para la parte de entrada/salida, también se encontraban agrupadas algunas funciones que no estaban del todo relacionadas entre sí. A la vez, se prevé necesario añadir nuevas funcionalidades relacionadas con la comunicación por el puerto serie para integrar el dispositivo con la interfaz gráfica, así como para solucionar los problemas detectados en el *rotary encoder*.

Es por ello que se ha decidido dividir también el código de esta parte en 3 archivos: uno para el puerto serie, otro para el codificador rotatorio, y un tercero para el manejo de eventos.

Especificaciones acerca de la comunicación por el puerto serie

Para realizar la comunicación con la interfaz gráfica, se necesitará definir un formato para los mensajes que intercambie esta con el dispositivo.

En primer lugar, conviene determinar de alguna forma el comienzo y el final de un mensaje, lo que nos permitirá evitar errores procesando datos indebidos. Para el comienzo común servirá cualquier carácter que no se vaya

a usar en ningún otro punto del mensaje. En este caso, se ha elegido el acento circunflejo (^). Para el final, bastará con enviar un retorno de línea (\n).

A continuación, se definirá un formato general para los mensajes. Como en este caso se va a realizar la comunicación por el puerto serie, que no es especialmente rápido, se priorizará que los mensajes sean lo más cortos posibles. De esta forma, llegamos a la siguiente especificación:

$$\text{^T,C(,P1,P2 \dots ,Pn)\n}$$

Donde:

- T representa el tipo de mensaje, indicando ? una petición y ! un envío de datos.
- C será una de las siguientes opciones, dependiendo del contenido que se esté transmitiendo:
 - @ servirá como saludo inicial para establecer la conexión entre ambas partes.
 - i para información general del dispositivo.
 - c para la contraseña.
 - n se usará antes de enviar ranuras de memoria, para indicar cuántas.
 - s para ranuras (Slots) de memoria.
 - p para señales PWM.
- Px serán los distintos parámetros, que dependerán del comando enviado y se encuentran documentados en el código.

Por último, cabría plantear el uso de algún mecanismo de detección de errores. En este caso, dada la baja complejidad y urgencia de la transmisión, se ha optado por posponer este aspecto del diseño, a la espera de probar la comunicación y determinar de forma experimental la frecuencia de errores.

4.2.3. Memoria EEPROM

Para el manejo de la memoria EEPROM, cabe especificar qué datos será necesarios guardar, así como la forma en la que se van a almacenar los mismos.

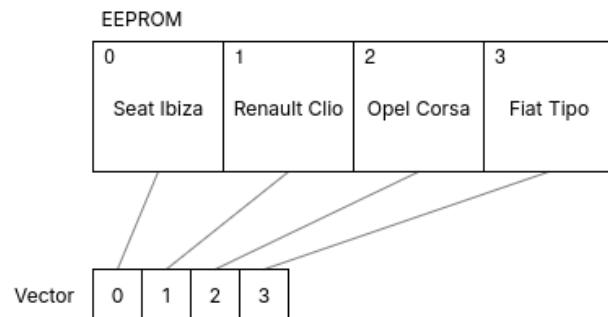
En primer lugar se definirán las siguientes unidades:

- **PWM:** Representación de una señal PWM en la memoria. Consistirá de sus 4 parámetros principales (modo, frecuencia, ciclo y fase), así como un nombre identificativo (por ejemplo, “Intermitentes”).
- **Slot:** Llamado así por ser la unidad en la que se compartimentará la memoria (ranuras), representa una forma de agrupar señales PWM. Permitirá al usuario definir distintos modelos de coches de forma que cada PWM esté destinado a probar un grupo distinto de luces de faros. Consistirá de 8 señales PWM y de un nombre identificativo. Se añadirá, además, una variable que indicará si la ranura está ocupada o no. Esto permitirá evitar borrados reales de memoria, prolongando su vida útil y disminuyendo el coste en cuanto a rendimiento de la escritura.

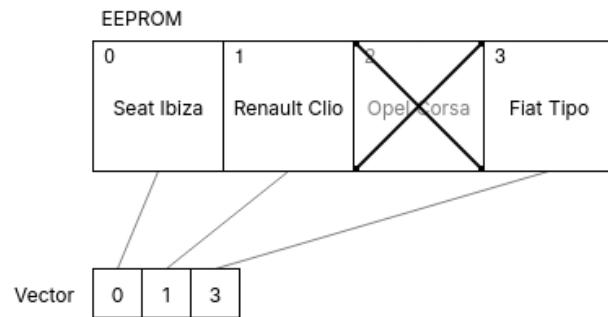
Otros parámetros que se necesitarán guardar, teniendo en cuenta los requisitos establecidos son:

- **Valor de inicialización:** Servirá para determinar si la memoria contiene o no datos, ya que en caso de que no los contenga, necesitará ser inicializada con algunos valores por defecto.
- **Número de serie:** Número identificador del dispositivo.
- **Versión del hardware.**
- **Versión del software.**
- **Brillo de la pantalla:** Permitirá mantener el brillo establecido entre ejecuciones.
- **Ranura por defecto:** Indica la ranura de memoria que se cargará automáticamente al iniciar el PWM Box.
- **Contraseña:** Debido al entorno en el que se plantea usar el dispositivo y al nivel de conocimiento técnico de sus usuarios, descrito en el anterior [análisis](#), no se estima necesario encriptar la contraseña de ninguna forma.

Dada la capacidad del usuario de añadir, borrar y reemplazar ranuras de memoria según desee, se prevé la necesidad de algún mecanismo adicional para determinar qué ranuras mostrar en el menú principal del dispositivo. Con este fin, se incluirá también un vector auxiliar, que contendrá el índice de las ranuras de la memoria en el orden en el que se les muestra al usuario. Actúa, por así decirlo, como un traductor entre el índice de las ranuras visibles al usuario y el índice de las mismas internamente. Esto puede entenderse más fácilmente con una representación, que se incluye en la figura 4.1



Usuario borra la ranura 2:



Usuario añade una ranura nueva:

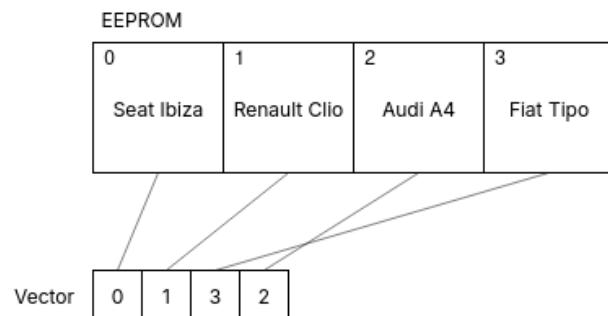


Figura 4.1: Representación del funcionamiento del vector auxiliar al realizar distintas operaciones sobre la EEPROM.

4.2.4. General

La organización general de los archivos dentro del proyecto también se considera un aspecto importante a tener en cuenta para facilitar su manejo. Tras los cambios planteados, algunos de los cuales incluyen la creación de aún más ficheros, esta necesidad se hace aún más aparente.

Para tratarlo, se plantea agruparlos en distintos directorios, a los que llamaremos módulos, de forma que queden distribuidos como se muestra en la figura 4.2.

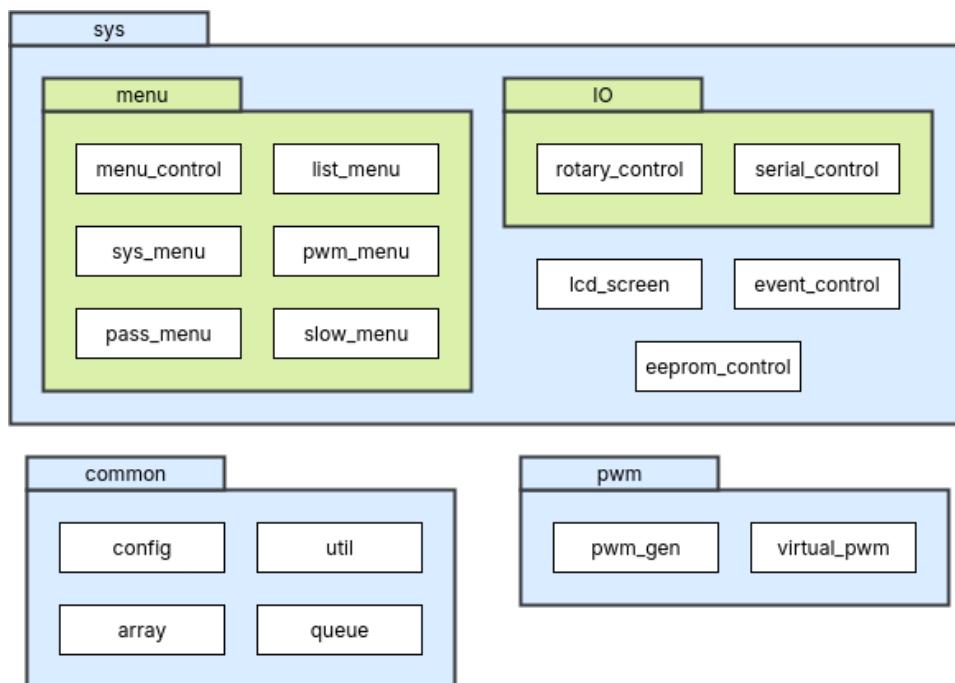


Figura 4.2: Distribución del código en distintos archivos y módulos.

- **Módulo del sistema:** Contiene la implementación de características claves para el funcionamiento del sistema.
 - **Módulo de E/S:** En él encontramos los archivos que definen el funcionamiento del *rotary encoder* y de la UART, responsables de la comunicación con el usuario y la aplicación respectivamente.
 - **Módulo de menús:** Agrupa la implementación de los distintos menús que usa el sistema.
- **Módulo de PWM:** Se encarga de la generación y configuración de las señales de salida.

- **Módulo de archivos comunes:** Se tratan de utilidades genéricas, pensadas para ser usadas en común por las distintas partes del programa.

De esta forma, los distintos archivos quedan más agrupados según su funcionalidad, haciendo más viable establecer cierta encapsulación del código. Esto es especialmente importante teniendo en cuenta que el lenguaje con el que se trata no se especializa en la programación orientada a objetos, por lo que no permite definir clases al contrario que su sucesor. Podemos ver cómo quedaría la ejecución del bucle principal en la representación de la figura 4.3. Adicionalmente, la figura 4.4 muestra el manejo de las interrupciones.

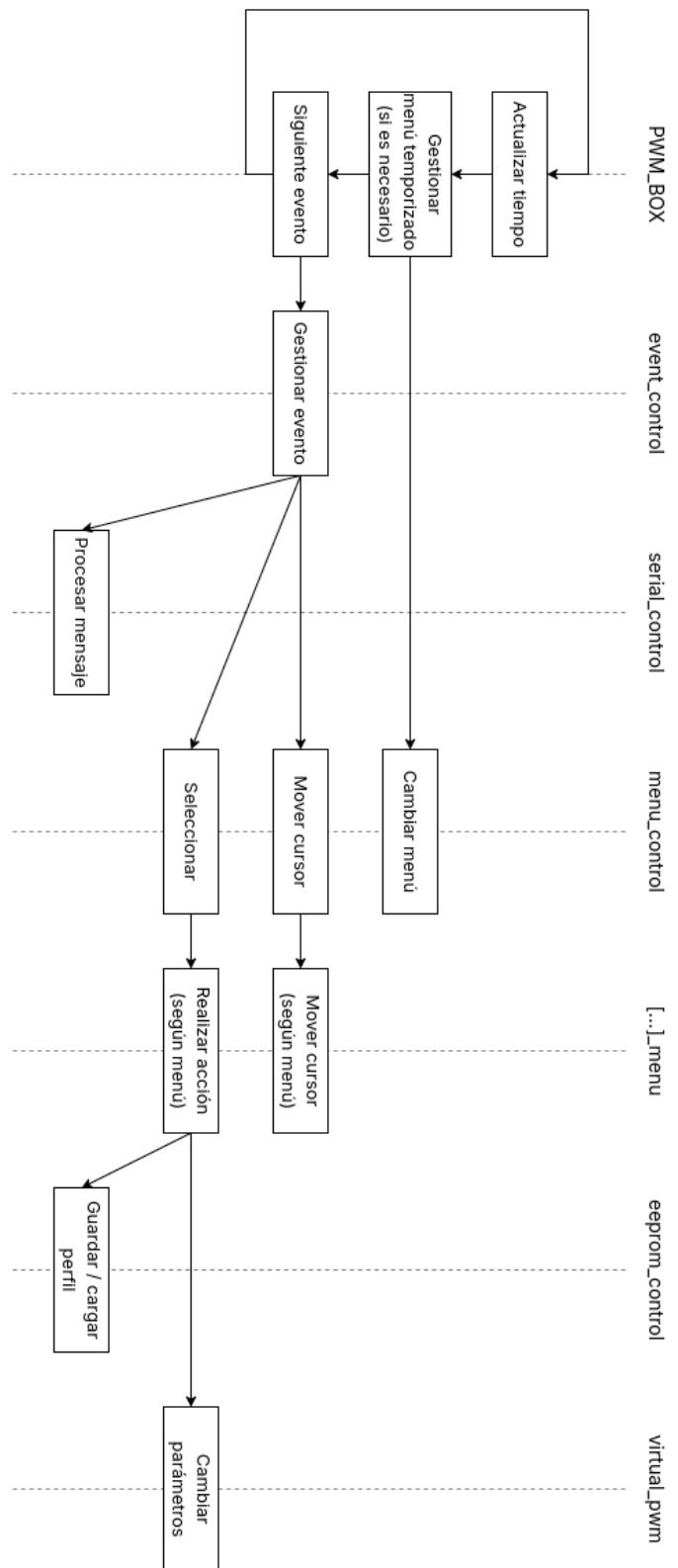


Figura 4.3: Ejecución del bucle principal.

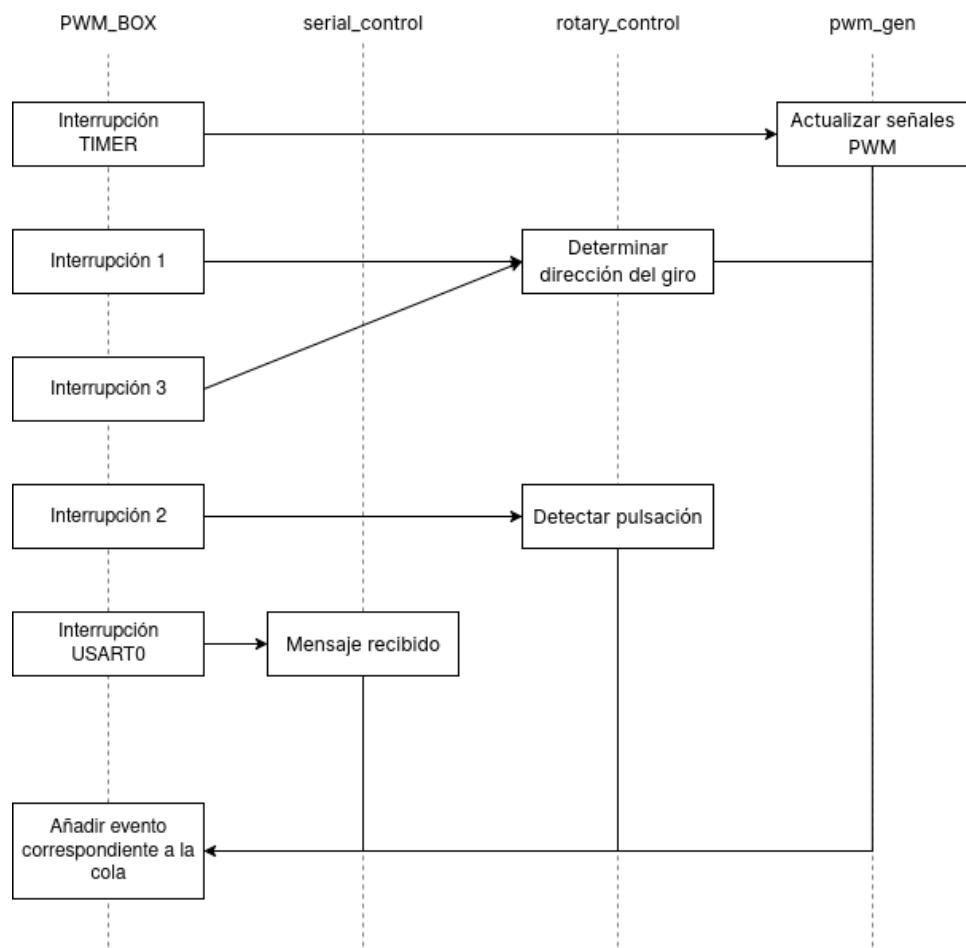


Figura 4.4: Ejecución de las distintas interrupciones.

4.3. Desarrollo

En esta sección se detallarán las decisiones enfrentadas durante la implementación del firmware, centrándose la atención en los requerimientos definidos.

Cabe mencionar, como regla general, que la mayoría sino todos los apartados incluidos en esta sección hacen uso de una forma u otra del *datasheet* del ATmega 2560 [8], microcontrolador incluido en el Arduino Mega 2560 del que se hace uso en este proyecto.

4.3.1. Menú de señales lentes

La funcionalidad de este menú se basa en un contador ejecutado en el bucle principal, que se incrementa cada medio segundo. Este es el máximo común divisor de la frecuencia con la que se actualizan estas señales. Cuando el usuario escoge una secuencia, este contador comienza a incrementarse, y a compararse en cada unidad con los instantes en los que la señal debe cambiar, los cuales se encuentran definidos en un *switch*. Esto permite cambiar la señal a una frecuencia menor de la que el dispositivo soporta de base, pudiendo realizar pruebas en faros que así lo requieran.

Para que este menú sea independiente del resto del sistema, se ha implementado en el *rotary encoder* la capacidad de detectar cuándo el botón ha sido mantenido (más detalles en la siguiente sección). Cuando esto ocurra, el sistema mostrará esta pantalla.

4.3.2. *Rotary encoder*

En este ámbito, el objetivo principal era el de corregir el funcionamiento algo errático del *rotary encoder*. Sin embargo, la lógica del código presente no estaba del todo clara, por lo que se acabó reimplementando desde cero. Esto también ha permitido incluir la posibilidad de detectar pulsaciones largas del botón, como se ha mencionado en el apartado anterior.

Pulsaciones

El pin correspondiente del microcontrolador está programado para generar una interrupción en cada cambio de flanco. Esto, en el caso del botón, quiere decir que se generará una interrupción al pulsarse el mismo y otra al soltarse. En cada una, habrá por lo tanto que comprobar el estado del botón en ese instante. En caso de que se encuentre pulsado, se guardará en una variable ese instante de tiempo. Si por el contrario no está pulsado, se comparará la variable anterior con el instante actual (momento en el que

termina la pulsación). De esta forma, se determina si el botón se ha pulsado o si se ha mantenido. Este será el resultado de la operación, cuyo evento correspondiente se añadirá al buffer en la rutina de la interrupción.

A esto se le añade una pequeña lógica de *debouncing*, que sólo tendrá en cuenta cada pulsación si la anterior se ha realizado fuera de una breve ventana de tiempo. Esto evita la detección de pulsaciones duplicadas debido a inexactitudes en el circuito del codificador.

Giros

La lógica para el giro del *rotary encoder* es más compleja. Debido al funcionamiento del mismo, la dirección de giro se puede determinar según **el orden** en el que dos pines emitan voltaje. De esta forma, determinando la secuencia de estados por los que pasan los pines al realizar giros para las dos direcciones, se puede crear una máquina de estados. Esta, comprobando el estado actual de los pines al manejarse la interrupción, determinará cuándo se ha completado una rotación. Al igual que en el caso anterior, cuando esto ocurra, se incluirá el evento correspondiente en la cola de eventos, que será procesado por el sistema.

Para seguir este método, también se programa el microcontrolador para que emita una interrupción en ambos flancos de ambos pines, y se va comparando su estado con el de la máquina de estados mencionada. Este funcionamiento se ha encontrado en la librería <https://github.com/buxtronix/arduino/tree/master/libraries/Rotary>, cuyo código fuente ha sido adaptado para funcionar en este proyecto.

4.3.3. Serial control

Una de los factores más importantes para la comunicación del puerto serie es configurar correctamente los puertos del microcontrolador. En este caso, se han habilitado las interrupciones RX (para la recepción) y TX (para el envío) y se ha configurado la comunicación para usar 8-bits por segmento.

Una de las desventajas de usar el puerto seria para la comunicación, es que sus velocidades de transmisión no son especialmente altas. Por ello, se plantea activar el modo *Double Speed Operation* del ATmega 2560 para llegar a 115200 baudios. Con este, el valor a establecer en el puerto UBBR vendrá dado por la siguiente ecuación: [8]

$$UBBR = \frac{f_{OSC}}{8 \cdot BAUD} - 1$$

Habiendo configurado el microcontrolador, el resto de la implementación se basa en definir funciones básicas que accedan al registro UDR0. Posteriormente, se usan estas funciones para implementar el sistema de envío y recepción de datos, ya cubiertos en la 4.2.

4.3.4. Memoria EEPROM

Esta parte se ha implementado utilizando el apoyo de la librería `avr/eeprom.h`, que proporciona la directiva `EEMEM`, la cual permite declarar variables en la memoria EEPROM. Estas, sin embargo, solo sirven para poder hacer referencia a la dirección de memoria en la que se encuentran los datos desde el programa. Para leer su valor o escribir en ellas, es necesario usar las funciones proporcionadas por la misma librería. Por ello, la mayoría de funciones incluidas en esta parte no son más que distintos *getters* y *setters* para las distintas variables determinadas en la sección 4.2.

A estas se une el vector auxiliar también mencionado en la sección 4.2. Para facilitar su manejo y hacer el código más reutilizable, se ha implementado una pequeña librería que define las funciones básicas esperables de un vector.

Con todo esto, el contenido de la memoria EEPROM puede verse en la figura 4.5.

```
typedef struct eeprom_t {
    uint8_t init_val;
    uint16_t serial;
    int8_t password[3];
    int8_t default_slot;
    uint8_t brightness;
    slot_t slots[NUM_SLOTS];
    array_t used_slots;
} eeprom_t;
```

Figura 4.5: Contenido de la memoria EEPROM.

Cuando, más adelante en el desarrollo, se comenzó a trabajar en la interfaz, se detectó un gran problema con el rendimiento a la hora de obtener los datos del dispositivo. El causante era que se estaban leyendo los datos directamente de la EEPROM siempre que la interfaz los pedía. Cada tiempo de acceso a un dato en memoria supone una gran penalización en el rendimiento del programa, que se hizo aparente incluso al leer únicamente la

contraseña. Si se mantenía este funcionamiento, la calidad del producto final se vería muy perjudicada.

Es por ello que hubo que tomar la decisión de mantener en todo momento una copia de los datos de la EEPROM en la memoria del programa.

Por un lado, esto soluciona el problema comentado, ya que leer datos de la RAM es mucho más rápido que el caso anterior. Además, puede evitar escrituras innecesarias en la EEPROM, comparando el valor que se pretende guardar con el presente para detectar si realmente ha habido un cambio (también posible anteriormente, pero con la penalización del tiempo de acceso). Esto supone una mayor vida útil de la memoria programable.

Sin embargo, dado que la directiva `EEMEM` ya reserva espacio para las variables en la memoria RAM, esto provoca en la práctica que las variables de la memoria EEPROM ocupen el doble de espacio. Dado que la memoria RAM del Arduino Mega 2560 tiene el doble de tamaño que su EEPROM, esto solo reduce en dos el número de perfiles que podemos almacenar, lo cual se considera un precio justo a pagar para cumplir el [RNF-2](#).

4.4. Pruebas

Conforme se ha ido completando el desarrollo de los distintos módulos, sus funcionalidades han sido validadas a través de pruebas unitarias, de integración y finalmente de aceptación.

Debido a la naturaleza del trabajo, la mayoría de estas pruebas se han realizado de manera experimental, observando la reacción del dispositivo a las acciones del usuario y comprobando que el resultado de las mismas es el esperado.

Para determinar que no se ha alterado la capacidad del dispositivo de emitir las señales de manera correcta, así como validar la implementación de las señales lentas, se ha usado un analizador lógico como comprobación adicional. En la figura 4.6 se muestran las señales PWM resultantes de distintas configuraciones. Adicionalmente, en las figuras 4.7 y 4.8, se ven dos de las secuencias de señales lentas incluidas con el dispositivo.

Estas pruebas han sido realizadas con el Saleae Logic v1.2.40 [10], que es software propietario. Sin embargo, se ha comprobado usando Sigrok PulseView v0.4.2 [12], una alternativa *open-source*, y el resultado ha sido idéntico.

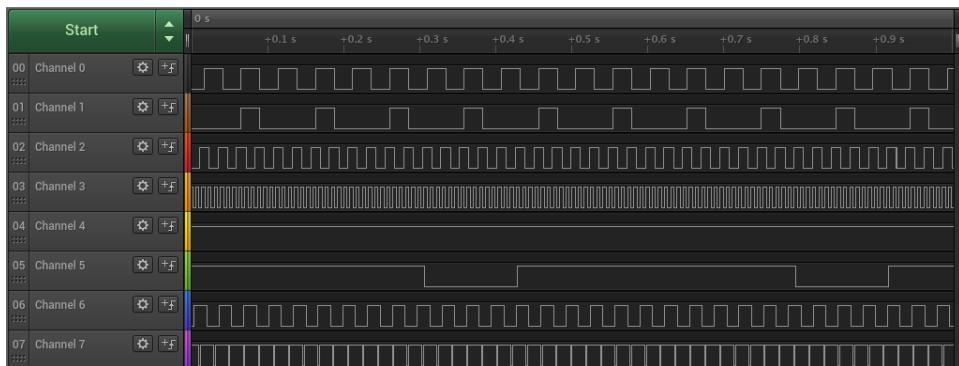


Figura 4.6: Señales PWM con distintas configuraciones.

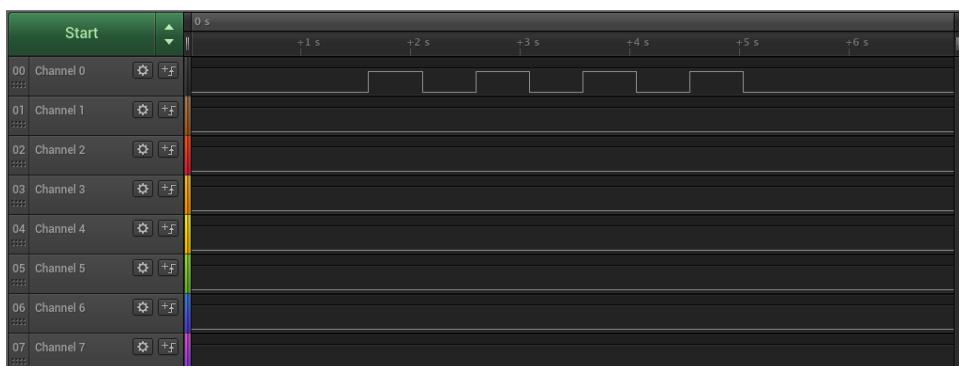


Figura 4.7: Una de las secuencias de señales lentas pre-configuradas.



Figura 4.8: Otra de las secuencias de señales lentas pre-configuradas.

Capítulo 5

Interfaz gráfica

5.1. Análisis

En esta sección se comenzará a trabajar en la aplicación, realizando en primer lugar un breve análisis para concretar sus requisitos.

En el caso de la interfaz gráfica, el ámbito de uso difiere un poco del concretado para el firmware en la sección 4.1. Esta se pretende usar desde Valeo para incluir en el dispositivo las ranuras que cada fabricante necesite antes de proporcionarles el PWM Box. Por lo tanto, se deduce que su uso va a ser en un entorno algo más administrativo que en el caso anterior.

Si bien se anticipa, por lo tanto, que el usuario tenga un mayor nivel de entendimiento a la hora de manejar sistemas informáticos, sigue sin esperarse ningún tipo de conocimiento técnico. Solamente se asumirá algo de destreza en el uso de programas de ofimática y similares.

Teniendo esto en cuenta, se determinarán los requisitos de la aplicación.

Comunicación con el dispositivo

Cuadro 5.1: RF-1. Conectar con el dispositivo.

ID	RF-1
Nombre	Conectar con el dispositivo
Descripción	El sistema debe ser capaz de encontrar el puerto del dispositivo en el equipo y conectarse a él de forma automática, tanto desde Windows como desde Linux.
Prioridad	Alta

Cuadro 5.2: RF-2. Obtener la información básica del dispositivo.

ID	RF-2
Nombre	Obtener la información básica del dispositivo
Descripción	El sistema debe poder obtener la información básica del dispositivo. Esto incluye su número de serie, su versión de <i>hardware</i> y su versión de <i>software</i> .
Prioridad	Alta

Cuadro 5.3: RF-3. Obtener la configuración general del dispositivo.

ID	RF-3
Nombre	Obtener la configuración general del dispositivo
Descripción	El sistema debe tener la capacidad de obtener la configuración general del dispositivo, que consiste de su contraseña y la ranura por defecto establecida.
Prioridad	Alta

Cuadro 5.4: RF-4. Obtener las ranuras guardadas en el dispositivo.

ID	RF-4
Nombre	Obtener las ranuras guardadas en el dispositivo
Descripción	El sistema debe comunicarse con el dispositivo para obtener las ranuras que estén guardadas en la memoria del mismo, incluyendo la información relativa a las señales PWM que las componen.
Prioridad	Alta

Cuadro 5.5: RF-5. Enviar configuración general al dispositivo.

ID	RF-5
Nombre	Enviar configuración general al dispositivo
Descripción	El sistema debe ser capaz de enviar de vuelta al dispositivo su configuración general.
Prioridad	Alta

Cuadro 5.6: RF-6. Enviar ranuras al dispositivo.

ID	RF-6
Nombre	Enviar ranuras al dispositivo
Descripción	El sistema debe poder enviar ranuras configuradas al dispositivo.
Prioridad	Alta

Gestión del dispositivo

Cuadro 5.7: RF-7. Mostrar la información básica del dispositivo.

ID	RF-7
Nombre	Mostrar la información básica del dispositivo
Descripción	El sistema debe permitir al usuario consultar la información básica del dispositivo.
Prioridad	Alta

Cuadro 5.8: RF-8. Mostrar la configuración general del dispositivo.

ID	RF-8
Nombre	Mostrar la configuración general del dispositivo
Descripción	El sistema debe permitir al usuario visualizar la configuración general del dispositivo.
Prioridad	Alta

Cuadro 5.9: RF-9. Mostrar los parámetros de las señales generadas.

ID	RF-9
Nombre	Mostrar los parámetros de las señales generadas
Descripción	El sistema debe mostrar al usuario los parámetros de las distintas señales PWM configuradas.
Prioridad	Alta

Gestión de ranuras

Cuadro 5.10: RF-10. Crear ranuras nuevas.

ID	RF-10
Nombre	Crear ranuras nuevas
Descripción	Crear ranuras nuevas.
Prioridad	Alta

Cuadro 5.11: RF-11. Exportar ranuras al equipo.

ID	RF-11
Nombre	Exportar ranuras al equipo
Descripción	El sistema debe permitir exportar las ranuras cargadas a la memoria del equipo en un formato adecuado.
Prioridad	Alta

Cuadro 5.12: RF-12. Importar ranuras del equipo.

ID	RF-12
Nombre	Importar ranuras del equipo
Descripción	El sistema debe permitir al usuario importar las ranuras almacenadas con un determinado formato en la memoria del equipo.
Prioridad	Alta

Cuadro 5.13: RF-13. Visualizar la configuración de las ranuras.

ID	RF-13
Nombre	Visualizar la configuración de las ranuras
Descripción	El sistema permitir al usuario consultar los parámetros de las señales PWM que componen las ranuras cargadas.
Prioridad	Alta

Cuadro 5.14: RF-14. Modificar los parámetros de las ranuras cargadas.

ID	RF-14
Nombre	Modificar los parámetros de las ranuras cargadas
Descripción	El sistema debe permitir la creación de ranuras nuevas desde la propia aplicación.
Prioridad	Alta

Requisitos no funcionales

Cuadro 5.15: RNF-1. Manejabilidad.

ID	RNF-1
Nombre	Manejabilidad
Descripción	El sistema ha de ser sencillo e intuitivo para los usuarios.
Prioridad	Alta

Cuadro 5.16: RNF-2. Rendimiento.

ID	RNF-2
Nombre	Rendimiento
Descripción	El sistema debe ser fluido y responder a las acciones del usuario sin demoras excesivas.
Prioridad	Alta

Cuadro 5.17: RNF-3. Aspecto atractivo.

ID	RNF-3
Nombre	Aspecto atractivo
Descripción	El sistema debe ser atractivo visualmente.
Prioridad	Media

5.2. Diseño

La fase de diseño de la interfaz comienza con un prototipado visual de la aplicación. Teniendo en cuenta el análisis realizado, se plantea mantener un aspecto simple, estando la información acerca de las distintas señales en una misma ventana. De esta forma, todos los aspectos de interés se encontrarán a simple vista, evitando que el usuario pierda tiempo innecesariamente para encontrar la información que busca. Esto se considera especialmente apropiado para el entorno al que se destina el producto.

Las distintas operaciones adicionales, como enviar perfiles al dispositivo o exportar archivos, sí se realizarán en ventanas separadas. Aún así, la complejidad de las mismas se mantendrá al mínimo, haciéndolas lo más intuitivas posible.

Con todo ello, el primer prototipo al que se llega puede verse en la figura 5.1. Sobre este se realizan numerosas iteraciones, centrándonos primero en incluir todas las funcionalidades necesarias, y posteriormente, en mejorar el aspecto, manejabilidad y accesibilidad de la interfaz en general. En las figuras 5.2 y 5.3 se pueden ver dos ejemplos de este proceso, hasta llegar a la versión final, mostrada en las figuras 5.2 y 5.5.

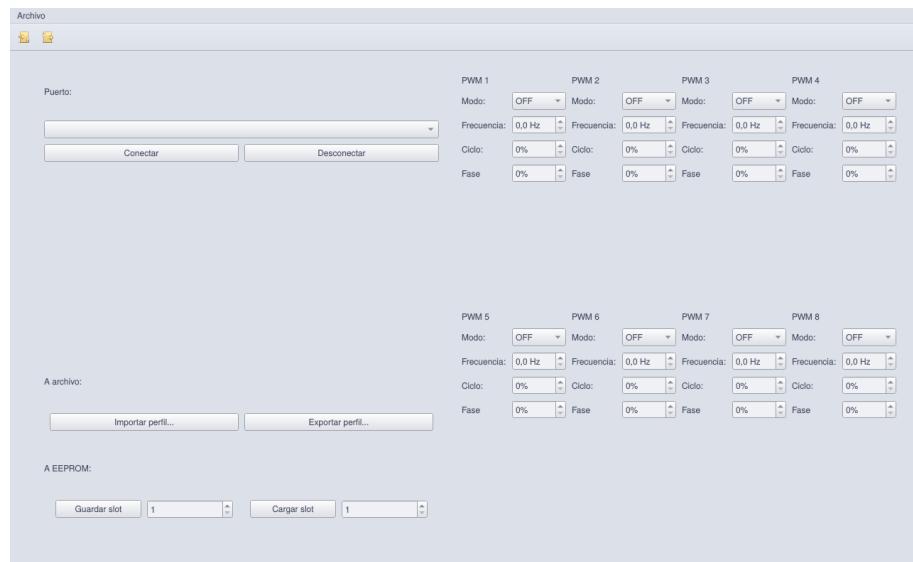


Figura 5.1: Primera iteración del diseño de la interfaz.

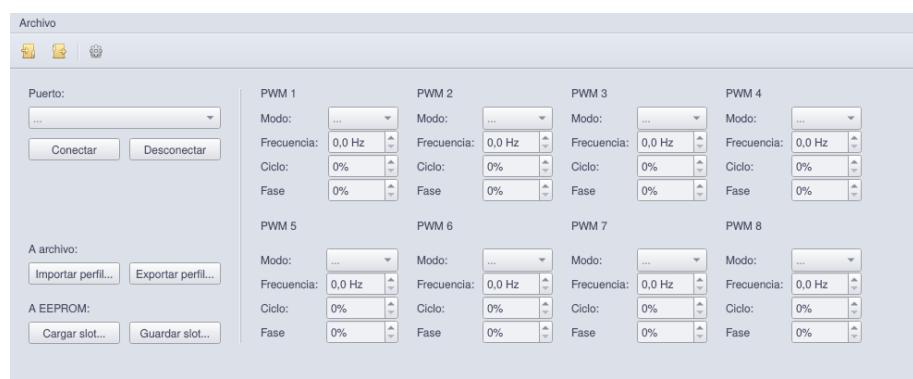


Figura 5.2: Otra iteración del diseño de la interfaz.

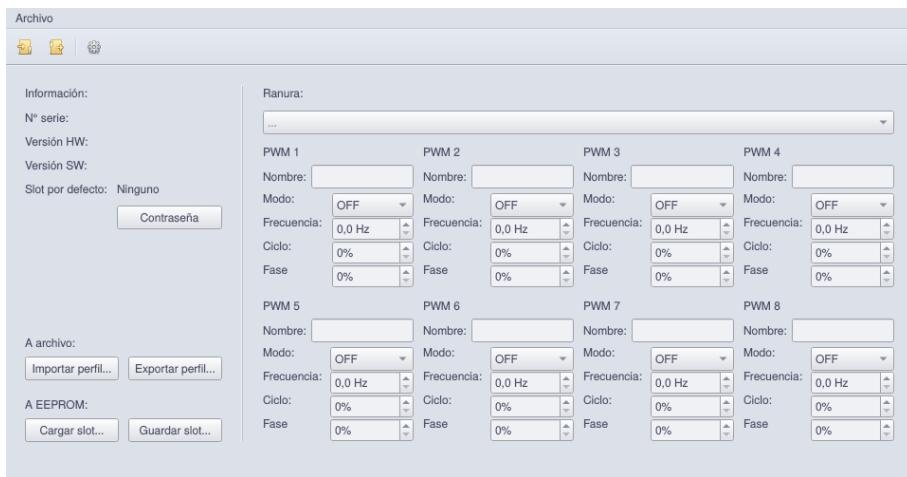


Figura 5.3: Otra iteración del diseño de la interfaz.

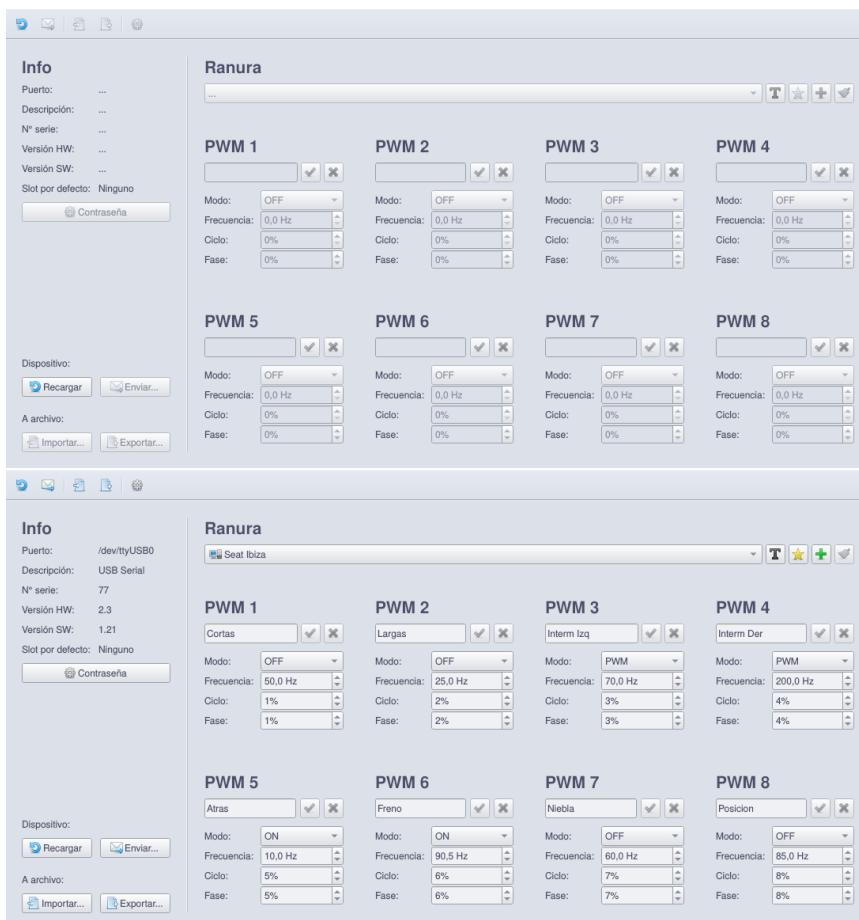


Figura 5.4: Diseño final de la interfaz.

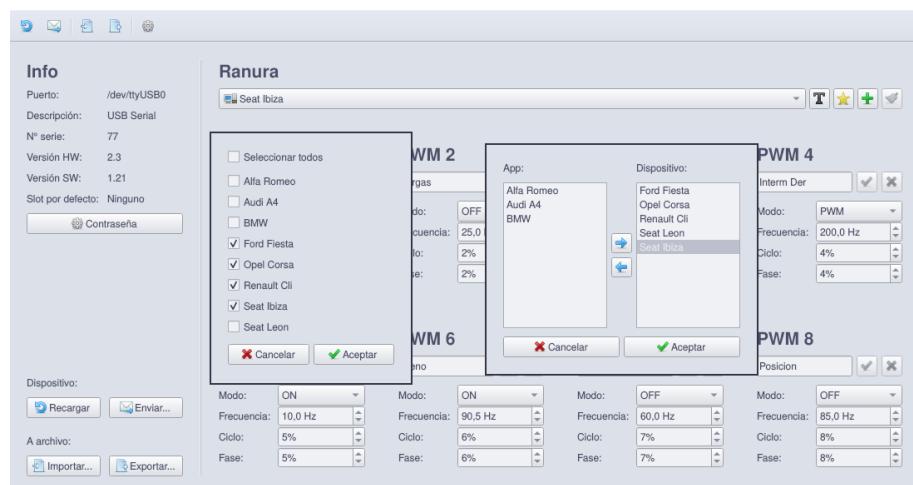


Figura 5.5: A la izquierda, ventana para exportar ranuras. A la derecha, ventana para enviar ranuras al dispositivo

En cuanto al diseño del *software* del sistema, la idea principal ha sido mantener la estructura general de los conceptos definidos para el dispositivo, con el objetivo de asegurar la compatibilidad de ambos desarrollos sin añadir complejidad innecesaria a la comunicación entre ambos. Los distintos módulos definidos se describen a continuación.

5.2.1. PWM Types

El primer aspecto a definir es la forma en la que se van a representar los PWM en esta parte del proyecto. Para ello, se crea el módulo PWM Types, cuyos objetos servirán como almacén de datos para las distintas señales PWM y perfiles cargados en la aplicación.

Un objeto de la clase PWM estará compuesto de la misma forma en la que se guardan las señales en la memoria del dispositivo (nombre, modo, frecuencia, ciclo, fase), pues estos son los parámetros externos que el usuario ha de ser capaz de modificar. De forma análoga, un objeto de la clase Slot contendrá su nombre y ocho objetos de la clase PWM.

5.2.2. JSON Manager

Para satisfacer los requisitos RF-11 y RF-12, se plantea el uso de archivos JSON. Estos resultan una forma sencilla de almacenar datos estructurados en un formato legible, la cual se adecúa a la jerarquía establecida entre un perfil y sus señales.

5.2.3. PWM Box

Adicionalmente, se define el módulo PWM Box. Este servirá de unión entre las funcionalidades de la interfaz y las del dispositivo, realizando las labores de comunicación por el puerto serie.

De igual forma que el módulo PWM Types, su contenido será análogo al del dispositivo, conteniendo, de los datos que este almacena en memoria, aquellos a los que el usuario deba tener acceso.

5.2.4. Main

Por último, el módulo principal es el que controlará todas las características de la interfaz. Estará dividido en distintas clases, cada una de las cuales contendrá la funcionalidad de una ventana distinta. Se distinguen las siguientes:

- **Ventana principal:** En ella se mostrará toda la información referente a las distintas señales PWM. Se permitirá al usuario escoger qué perfil está visualizando, así como realizar las distintas operaciones definidas en los [requisitos](#).
- **Ventana de envío:** Permitirá al usuario elegir cuáles de los perfiles cargados en la aplicación serán enviados al dispositivo. Será una prioridad representar esta acción de la forma más intuitiva posible.
- **Ventana de exportación:** Servirá para seleccionar qué perfiles se van a exportar.
- **Ventana de contraseña:** Mostrará la contraseña del dispositivo, y permitirá cambiarla por una nueva. También se usará cuando se abra la aplicación sin haber establecido una contraseña anteriormente, para obligar al usuario a hacerlo.

También se plantea el uso de alguna ventana adicional para usar como pantalla de carga si resulta oportuno, en caso de detectar algún proceso que tarde en ejecutarse.

Otro aspecto a tener en cuenta antes de comenzar la implementación, es el flujo de ejecución general de la aplicación, especialmente en lo que a envío de datos al dispositivo se refiere. En un primer momento se distinguen dos opciones.

Opción 1: Envío de datos continuo

Se considera una forma más “moderna” de conceptualizar la aplicación. En ella, cualquier cambio realizado sobre los parámetros de las señales sería enviado automáticamente al dispositivo. De igual forma, las ranuras creadas se añadirían también de forma automática a este. Esto permitiría sustituir completamente los controles físicos del PWM Box.

Esta sería la opción adecuada para una aplicación más orientada al ocio, que se pretenda distribuir para el uso cotidiano de todo tipo de usuarios. En este caso la comodidad que este enfoque ofrece sería ventajosa.

Opción 2: Envío de datos controlado

En este modelo de funcionamiento, se prioriza el control del usuario sobre el dispositivo. Los cambios que se realicen en la aplicación tendrán que ser manualmente aplicados, asegurando que la forma en la que funcione el dispositivo en todo momento sea intencionada.

Este es el enfoque que se ha elegido finalmente para la interfaz, ya que se considera más apropiado para el contexto y la forma en el que se usará. De esta manera, la aplicación servirá más como un almacén de perfiles, que podrán ser cargados en un PWM Box cuando se necesiten. Esta opción se alinea más con las necesidades expresadas por Valeo.

5.2.5. General

Habiendo planteado los distintos módulos, las interacciones entre ellos pueden modelarse a través de los diagramas de secuencia que se muestran a continuación.

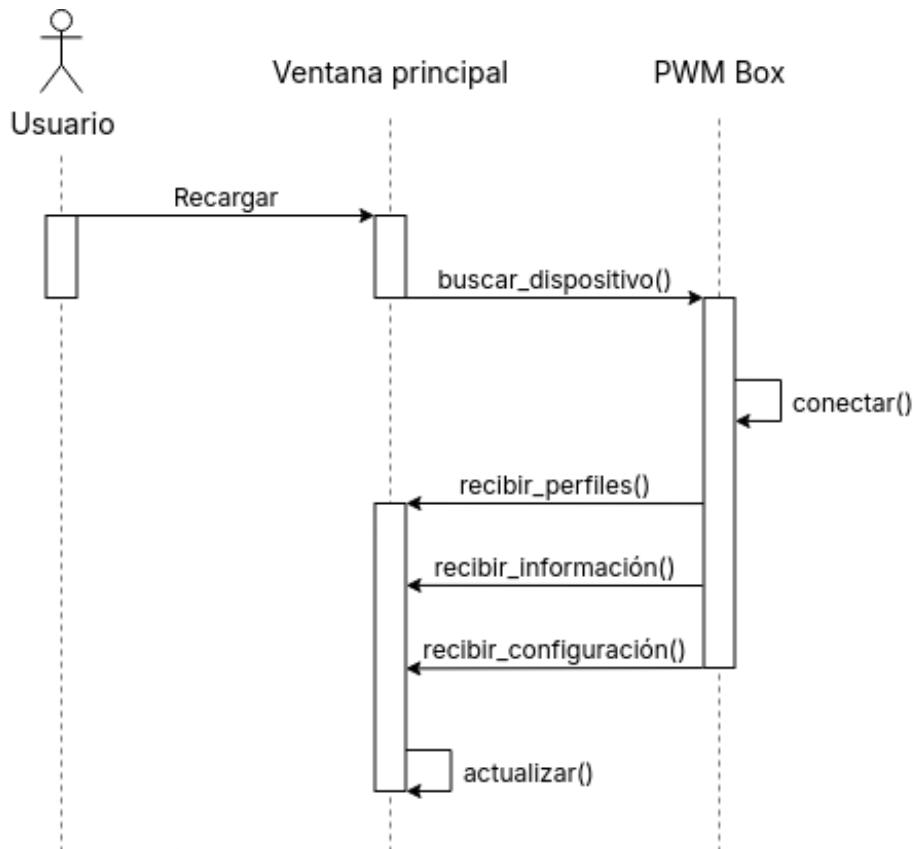


Figura 5.6: Flujo de ejecución al recargar la conexión con el dispositivo.

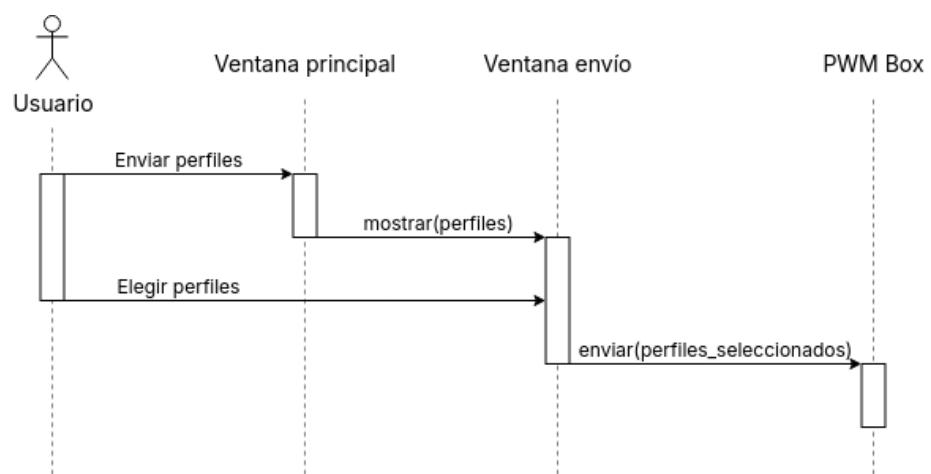


Figura 5.7: Flujo de ejecución al enviar perfiles al dispositivo.

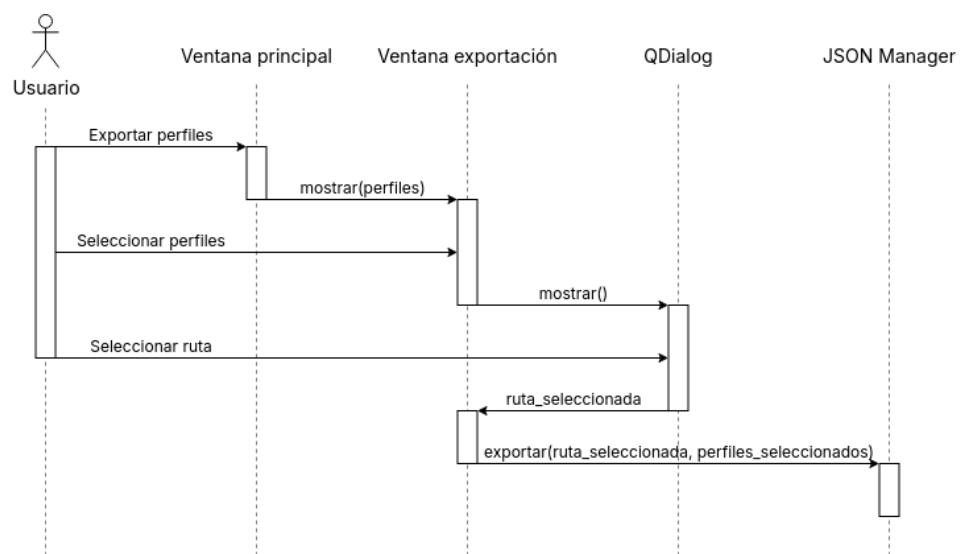


Figura 5.8: Flujo de ejecución al exportar perfiles al equipo.

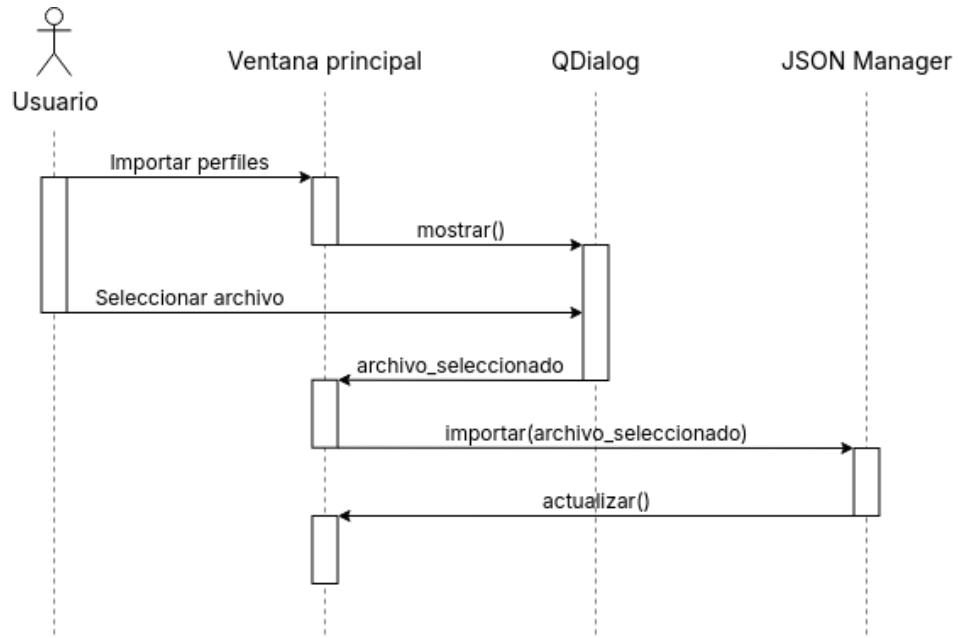


Figura 5.9: Flujo de ejecución al importar perfiles desde el equipo.

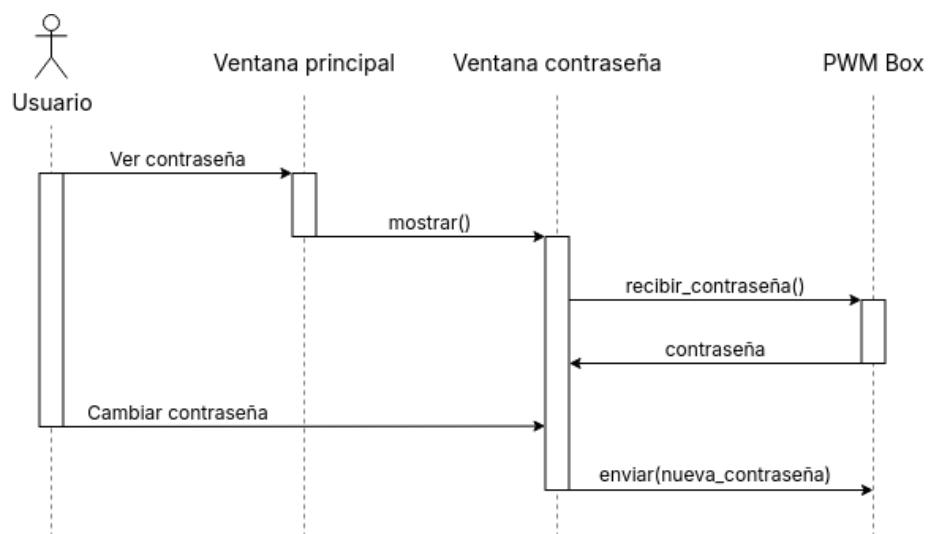


Figura 5.10: Flujo de ejecución al mostrar y modificar la contraseña.

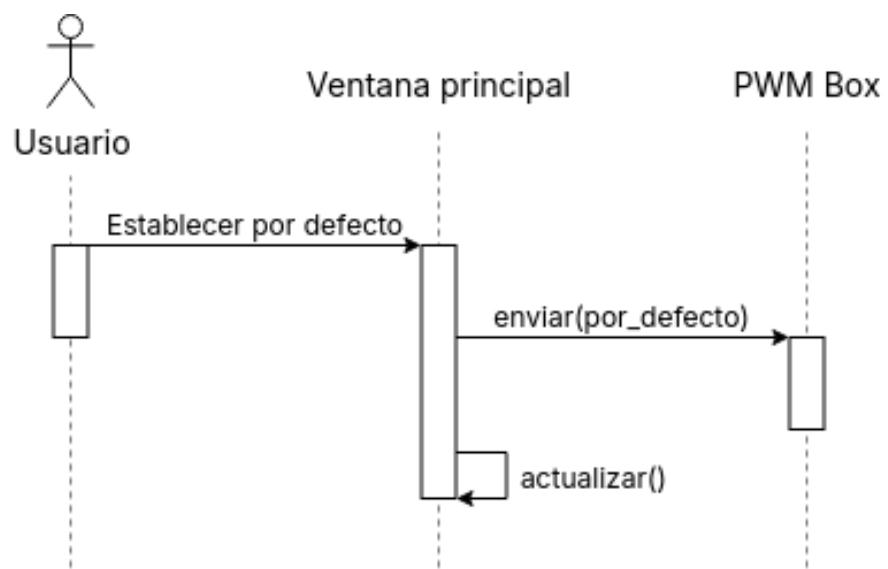


Figura 5.11: Flujo de ejecución al establecer el perfil por defecto.

5.3. Desarrollo

Para el desarrollo de la aplicación ha sido necesario el uso de la librería *pyserial* [9]. Las funcionalidades que proporciona han permitido gestionar todos los aspectos de la comunicación por el puerto serie.

También se ha hecho uso de la librería estándar *json* [5] para la codificación y decodificación de los archivos en este formato.

Por último, para la apariencia general de la aplicación, se ha usado el tema *catppuccin latte* de la librería *qt-themes* [2].

Durante el proceso de implementación se ha enfrentado una situación que no se había tenido en cuenta en la etapa de diseño: la forma de gestionar las distintas fuentes de las que puede provenir un perfil. Como se determinó en la definición de los **requisitos**, un perfil puede cargarse en la aplicación desde el dispositivo, desde un archivo importado, o puede ser directamente creado a través de la interfaz. Es conveniente manejar estas tres fuentes por separado, ya que cada una tiene ciertas peculiaridades. Por ejemplo, no tendría sentido permitir al usuario quitar de la aplicación un perfil que se encuentre cargado en el dispositivo, mientras que si lo tendría en el caso de tratarse de una ranura creada desde la interfaz.

Otro problema al que se ha hecho frente constituye una de las desventajas de la comunicación por el puerto serie: su velocidad. Se ha comprobado que al transmitir un número mayor de perfiles el proceso se alarga, llegando a bloquear momentáneamente el hilo de ejecución de la interfaz. Esto se debe a que el dispositivo necesita esperar entre cada envío de datos para recibirlas y procesarlos. Para intentar suavizarlo, se ha ajustado la recepción de datos del dispositivo, dejando el procesamiento más costoso para el final, una vez haya terminado la transmisión. Sin embargo, debido a la naturaleza de la adversidad, no se ha podido solucionar completamente.

Además, para evitar el bloqueo de la interfaz, se ha implementado la ejecución paralela de un hilo encargado de la comunicación por el puerto serie. Esto ha permitido también mostrar una ventana adicional, indicando al usuario que la transmisión de datos se encuentra en proceso.

5.4. Pruebas

En un primer lugar, se han ido llevando a cabo pruebas unitarias a lo largo del desarrollo, poniendo especial atención en los aspectos más transparentes al usuario (por ejemplo, el intercambio de datos con el dispositivo). Su objetivo ha sido evaluar el funcionamiento de los aspectos más esenciales, cuyos errores pueden resultar más difíciles de detectar en fases futuras del

desarrollo.

Adicionalmente, tras completar cada funcionalidad general (envío de perfiles, exportación de archivos, etc.), se han realizado pruebas de integración para comprobar la validez del flujo de las operaciones, tal y como se han definido en los [diagramas de secuencia](#).

Por último, cuando el desarrollo se ha completado, se han realizado pruebas de aceptación para asegurar el funcionamiento en conjunto de la aplicación, corrigiendo los pequeños errores encontrados.

Capítulo 6

Presupuesto

Con el objetivo de estimar el coste de producción del proyecto en un escenario real, se hará un listado de los distintos costes que sería necesario asumir.

Costes de material

En primer lugar, el equipo usado para este proyecto es un Lenovo T560, con un precio aproximado de 1400€. Se estima un ciclo de vida útil de unos 8 años, de los cuales alrededor de medio año se ha dedicado al desarrollo de este trabajo.

También sería necesario incluir el coste de los distintos componentes requeridos para la construcción del PWM Box. Los principales aparecen listados a continuación, pero también se ha añadido un sobrecoste para otros elementos como cables, los conectores, etc.

Por último, en cuanto al *software*, la mayoría de programas usados son completamente *open-source*. El único que sí tiene una versión comercial es Qt Designer, usado para el diseño de la interfaz gráfica. Su versión de código abierto se distribuye en su mayor parte bajo GPLv3. Esta licencia obliga a cualquier producto que use el *software* licenciado, incluso como librería, a ser distribuido también usando GPLv3 o GPLv3. Por lo tanto, si realmente se pretendiese distribuir el producto final, habría que determinar qué tipo de licencia usaríamos para saber si existe la necesidad de adquirir una licencia comercial o no. Por hacer el ejemplo más completo e informativo, nos pondremos en el peor de los casos, y asumiremos que pretendemos distribuir el producto final como software propietario. [3] [6]

Teniendo todo esto en cuenta, el coste desglosado sería el siguiente:

Equipo:	$\frac{1400\text{€}}{8 \text{ años}} \times 0.5 \text{ años} = 87.5\text{€}$
Arduino Mega 2560:	52.8€
Pantalla LCD 20x4:	10€
Rotary encoder:	4€
PCB:	20€
Carcasa:	50€
Otros elementos:	30€
Licencia Qt Designer:	$2270\text{€}/\text{año} \times 0.5 \text{ años} = 1135\text{€}$
Total:	1389.3€

Costes de personal

Estimamos que el sueldo de un programador junior en la actualidad ronda los 9.25€/hora. El de uno senior, por otro lado, se aproxima a unos 23€/hora. Como duración del proyecto, se tendrá en cuenta el **tiempo invertido real**, en concordancia con lo comentado en la **planificación temporal**.

Programador junior:	$9.25\text{€}/\text{hora} \times 450 \text{ horas} = 4162.5\text{€}$
Programador senior:	$23\text{€}/\text{hora} \times 2 \text{ horas} \times 4 \text{ semanas} \times 5 \text{ meses} = 920\text{€}$
Total:	5082.5€

Costes indirectos

Para hacer una aproximación, se tendrá en cuenta los siguientes gastos:

Luz:	$85\text{€}/\text{mes} \times 5 \text{ meses} = 425\text{€}$
Aqua:	$40\text{€}/\text{mes} \times 5 \text{ meses} = 200\text{€}$
Internet:	$25\text{€}/\text{mes} \times 5 \text{ meses} = 125\text{€}$
Total:	750€

Coste total

En base a esto, el coste total sería:

Costes de material:	2055.3€
Costes de personal:	5082.5€
Costes indirectos:	750€
Total:	7221.8€

Capítulo 7

Conclusiones y trabajos futuros

7.1. Conclusiones

Este proyecto perseguía el objetivo de implementar nuevas funcionalidades en un dispositivo testeador de faros de vehículos, así como desarrollar una interfaz gráfica que lo complemente.

Para ello, se ha implementado en el firmware del PWM Box la capacidad de almacenar perfiles de configuración en la memoria EEPROM del microcontrolador. También se ha desarrollado un nuevo menú, que permite comprobar el funcionamiento de faros que funcionen a una frecuencia menor de la que es capaz de generar el dispositivo usando señales PWM. Adicionalmente, se han corregido varios errores existentes que afectaban a la experiencia del usuario.

En cuanto a la interfaz, se ha usado el *framework* Qt en combinación con PySide6 para diseñar e implementar una aplicación multiplataforma que permita almacenar y enviar distintos perfiles al dispositivo. Además, se ha incluido la capacidad de exportar dichos perfiles en formato JSON, de forma que puedan ser importados de nuevo cuando se necesite.

Todo esto se ha llevado a cabo teniendo en cuenta los requisitos propuestos en las secciones 4.2 y 5.2 respectivamente. Por ello, se puede considerar que los objetivos marcados se han cumplido eficazmente.

En lo personal, estoy bastante satisfecho con el resultado final del proyecto. Me ha permitido poner en práctica muchos de los conocimientos adquiridos a lo largo del Grado, saliendo en ocasiones de mi zona de confort para obtener otros nuevos.

Si tuviera que mencionar un aspecto en el que considero que podría

haberlo hecho mejor, sería en la gestión del tiempo. En muchas ocasiones a lo largo del desarrollo, he dejado que mi opinión personal sobre qué estaba bien y qué estaba “menos bien” dictara la parte que merecía más trabajo. Esto me ha hecho pasar demasiado tiempo intentando perfeccionar algunas secciones que no lo necesitaban realmente. En el futuro trataré de evitar guiarme en exceso por mi juicio, y ceñirme mejor a la planificación inicial.

Esto no quita que, a la vez, me sienta orgulloso de haber sido capaz de compaginar este proyecto con un puesto a tiempo completo de 43 horas semanales, y aun así haber obtenido un resultado que considero positivo.

7.2. Trabajos futuros

Aunque los cambios realizados hayan sido suficientes para completar el objetivo actual, aún cabe la posibilidad de mejorar el producto en algunos aspectos. A continuación se comentan algunos de ellos, así como algunas posibles opciones que explorar para cada uno.

Memoria

En primer lugar, el número de ranuras que puede almacenar actualmente el dispositivo es limitado, permitiendo un máximo de sólo 15 perfiles. Como se desarrolló en la sección [4.3.4](#), optimizando el uso de memoria RAM del programa se podrían conseguir almacenar dos ranuras más. Sin embargo, esto conllevaría una gran penalización en el tiempo de acceso a los datos, por lo que no se considera una solución real.

Cabría también la posibilidad de plantear una gestión dinámica de la memoria, que limitase el tamaño de las variables y vectores de cada ranura según el valor o el número de elementos que estas contengan. No obstante, sería una solución muy compleja que, en caso de que las variables contengan valores cercanos a sus máximos, no tendría ningún efecto.

Una tercera solución, que si se cree más adecuada, sería desarrollar el almacenamiento de datos en una tarjeta SD. Esto requeriría una modificación del hardware para incluir el módulo correspondiente en el Arduino, pero resultaría una opción bastante interesante porque también permitiría el intercambio de información entre varios dispositivos PWM Box.

Comunicación

Esta posible ampliación de la memoria nos lleva a una siguiente mejora: la velocidad de transmisión de los datos. Actualmente, enviar el máximo número de ranuras posibles de la interfaz al dispositivo conlleva una espera

de alrededor de dos minutos. Esto se debe a las limitaciones de la comunicación por el puerto serie, que requiere una espera entre cada envío de datos para asegurar que el dispositivo los reciba correctamente. Si se amplía el número de ranuras, también se incrementa el tiempo de espera al trasmitir ese nuevo número máximo de ranuras al PWM Box.

Para solucionarlo, sería necesario plantear el uso de otro tipo de comunicación, o dado que el Arduino Mega 2560 tiene cuatro puertos serie, explorar la posibilidad de usar más de uno al mismo tiempo. No sería una solución simple, puesto que habría que mantener más de una conexión al mismo tiempo, y coordinarlas para que el orden final de los datos fuera siempre el adecuado, pero podría proponerse.

Interfaz

Por último, aunque la interfaz gráfica se pretende usar desde un ordenador, siempre cabe la posibilidad de aumentar la compatibilidad a un mayor número de sistemas. Para ello, se podría considerar una implementación usando otros *frameworks* como Flutter, Electron o NodeJS, que son muy ampliamente usados en la actualidad.

Bibliografía

- [1] avrdudes: *AVRDUDE*. <https://github.com/avrdudes/avrdude>, Accedido el 5 de Septiembre de 2025.
- [2] beatreichenbach: *qt-themes*. <https://github.com/beatreichenbach/qt-themes>, Accedido el 5 de Septiembre de 2025.
- [3] T. Q. Company: *Obligations of the GPL and LGPL*. <https://www.qt.io/licensing/open-source-lGPL-obligations>, Accedido el 5 de Septiembre de 2025.
- [4] B. Dennis M. Ritchie: *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 2nd ed., 1988, ISBN 9789688802052.
- [5] P. S. Foundation: *json*. <https://docs.python.org/3/library/json.html>, Accedido el 5 de Septiembre de 2025.
- [6] I. GitHub: *GNU Lesser General Public License v3.0*. <https://choosealicense.com/licenses/lgpl-3.0/>, Accedido el 5 de Septiembre de 2025.
- [7] KDE: *Plasma Desktop*. <https://kde.org/plasma-desktop/>, Accedido el 5 de Septiembre de 2025.
- [8] Microchip: *ATmega640/V-1280/V-1281/V-2560/V2561/V [DATASHEET]*, 2020. <https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/ATmega640-1280-1281-2560-2561-Datasheet-DS40002211A.pdf>, Rev. DS40002211A-05/2020.
- [9] pserial: *pyserial*. <https://github.com/pyserial/pyserial>, Accedido el 5 de Septiembre de 2025.
- [10] Saleae: *Saleae Logic 1.x*. <https://support.saleae.com/logic-software/legacy-software/older-software-releases>, Accedido el 5 de Septiembre de 2025.

- [11] I. J. Sang H. Son (ed.): *Handbook of Real-Time and Embedded Systems*, cap. Safe and Structured Used of Interrupts in Real-Time and Embedded Software, págs. 6–8. Chapman & Hall, 1st ed., 2007, ISBN 9781584886785.
- [12] Sigrok: *PulseView*. <https://sigrok.org/wiki/Downloads>, Accedido el 5 de Septiembre de 2025.

