



**UNIVERSIDAD
DE GRANADA**

**TRABAJO FIN DE GRADO
INGENIERÍA INFORMÁTICA**

Aplicación de control para testeador de faros de vehículos

Autor

Jose Manuel García Cazorla

Directores

Andrés María Roldán Aranda



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

—
Granada, Septiembre de 2025



Aplicación de control para testeador de faros de vehículos

Autor

Jose Manuel García Cazorla

Directores

Andrés María Roldán Aranda

Aplicación de control para testeador de faros de vehículos

Jose Manuel García Cazorla

Palabras clave: firmware, microcontrolador, Arduino, PWM, aplicación, frontend, 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 solo 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.

Control app for a vehicle headlight tester

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 Ingernierías 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 1 de Septiembre de 2025.

D. **Andrés María Roldán Aranda**, Profesor del área de TODO 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 1 de Septiembre de 2025.

El director:

Andrés María Roldán Aranda

Agradecimientos

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Conocimientos previos	3
1.3. Objetivos	3
1.4. Estructura del documento	4
2. Estado del arte	5
2.1. Ingeniería inversa	5
2.1.1. Archivo principal	6
2.1.2. EEPROM	7
2.1.3. Entrada/Salida	8
2.1.4. LCD	8
2.1.5. Vista general	9
2.2. Tecnologías usadas	10
2.2.1. Sistema operativo	10
2.2.2. Lenguajes de programación	10
2.2.3. Entorno de desarrollo	12
2.3. Planificación temporal	13
3. Firmware	15
3.1. Análisis	15
3.2. Diseño	18
3.2.1. Menús	18
3.2.2. Entrada/Salida	19
3.2.3. Memoria EEPROM	20
3.2.4. General	21
3.3. Desarrollo	26
3.3.1. Menú de señales lentas	26
3.3.2. <i>Rotary encoder</i>	26
3.3.3. Serial control	27
3.3.4. Memoria EEPROM	28
3.4. Pruebas	28

4. Interfaz gráfica	31
4.1. Análisis	31
4.2. Diseño	34
4.2.1. JSON Manager	35
4.2.2. PWM Types	35
4.2.3. PWM Box	35
4.2.4. Main Window	35
4.3. Desarrollo	35
4.4. Pruebas	35

Índice de figuras

1.1.	Dispositivo PWM Box desde arriba.	2
1.2.	Lateral del dispositivo PWM Box.	2
2.1.	Contenido de <i>PWM_BOX.RAR</i>	5
2.2.	Diagrama de Gantt que muestra la organización de las distintas fases del proyecto a lo largo de las semanas.	14
3.1.	Representación del funcionamiento del vector auxiliar al realizar distintas operaciones sobre la EEPROM.	22
3.2.	Distribución del código en distintos archivos y módulos. . . .	23
3.3.	Ejecución del bucle principal.	24
3.4.	Ejecución de las distintas interrupciones.	25
3.5.	Contenido de la memoria EEPROM.	28
3.6.	Señales PWM con distintas configuraciones.	29
3.7.	Una de las secuencias de señales lentas preconfiguradas. . . .	29
3.8.	Otra de las secuencias de señales lentas preconfiguradas. . . .	30

Índice de cuadros

3.1.	RF-1. Guardar perfiles.	16
3.2.	RF-2. Cargar perfiles.	16
3.3.	RF-3. Eliminar perfiles.	16
3.4.	RF-4. Guardar la contraseña.	17
3.5.	RF-5. Guardar el brillo de la pantalla.	17
3.6.	RF-6. Mostrar secuencias disponibles.	17
3.7.	RF-7. Ejecutar secuencias lentas.	17
3.8.	RF-8. Parar la ejecución de secuencias lentas.	18
4.1.	RF-1.1. Conectar con el dispositivo.	31
4.2.	RF-1.2. Obtener la información básica del dispositivo.	32
4.3.	RF-1.3. Obtener la configuración general del dispositivo.	32
4.4.	RF-1.4. Obtener las ranuras guardadas en el dispositivo.	32
4.5.	RF-1.5. Enviar configuración general al dispositivo.	32
4.6.	RF-1.6. Enviar ranuras al dispositivo.	32
4.7.	RF-2.1. Mostrar la información básica del dispositivo.	33
4.8.	RF-2.2. Mostrar la configuración general del dispositivo.	33
4.9.	RF-2.3. Mostrar la configuración general del dispositivo.	33
4.10.	RF-3.1. Crear ranuras nuevas.	33
4.11.	RF-3.2. Exportar ranuras al equipo.	34
4.12.	RF-3.3. Importar ranuras del equipo.	34
4.13.	RF-3.4. Visualizar la configuración de las ranuras.	34
4.14.	RF-3.5. Obtener las ranuras guardadas en el dispositivo.	34

Capítulo 1

Introducción

1.1. Motivación

Este trabajo se realiza a partir de la necesidad de Valeo, una empresa del ámbito automobilístico, de realizar pruebas más específicas sobre los faros que producen. El laboratorio de GranaSAT, del que mi tutor forma parte, fue el encargado de desarrollar las primeras versiones del dispositivo, que fueron anteriormente presentadas como Trabajos de Fin de Grado por Luis Sánchez (autor de la PCB y todo el apartado electrónico) y Rubén Sánchez (desarrollador de una parte del firmware), respectivamente.

Este dispositivo está compuesto por un Arduino Mega 2560, una pantalla LCD de 4 líneas, un *rotary encoder* y 8 conectores de salida. Por cada uno de estos 8 conectores se produce una salida PWM configurable por el usuario, pensada para conectar y probar los distintos modos del faro de un vehículo. En la Figura 1.1 incluída se puede ver la parte superior del dispositivo. Por otro lado, la Figura 1.2 muestra su lateral.



Figura 1.1: Dispositivo PWM Box desde arriba.



Figura 1.2: Lateral del dispositivo PWM Box.

1.2. Conocimientos previos

En la realización de este trabajo se han puesto en práctica competencias adquiridas en distintas asignaturas de la titulación, y más concretamente de la rama que he cursado, *Ingeniería de Computadores*. Por nombrar algunas:

- **Fundamentos y Metodología de la Programación:** Tratándose de un trabajo de desarrollo, se deben mencionar las asignaturas del Grado en las que se establecen los pilares del conocimiento en programación de muchos estudiantes. Desde lo más básico hasta conceptos más complejos, una gran parte de su contenido se ha aplicado en el desarrollo de este TFG.
- **Estructuras de Datos:** Como su nombre indica, se centra en los distintos contenedores de datos (vectores, colas, pilas, etc), sus particularidades, y cómo trabajar con ellos, conocimientos que han sido de gran utilidad a lo largo de este proyecto.
- **Sistemas con Microprocesadores:** Esta asignatura sirve de introducción a numerosos aspectos de la programación de microcontroladores, culminando en la construcción de un robot de lucha usando un Arduino. Todo ello ha sido puesto en práctica para desarrollar el firmware expuesto.
- **Sistemas Empotrados:** Complementa a la anterior, entrando más en detalle en la arquitectura de los microcontroladores, y aún a más bajo nivel. Aporta conocimientos que han sido de gran valor a lo largo de este proyecto.

1.3. Objetivos

El objetivo general de este TFG es mejorar el firmware de la versión existente del dispositivo, al que llamaremos *PWM Box*, corrigiendo algunos errores presentes en ella y añadiendo nuevas funcionalidades. Además, se plantea la implementación de una interfaz gráfica compatible con Windows y Linux, que permita el almacenamiento y gestión de distintos perfiles de configuración para el dispositivo.

A raíz de este, se han concretado otros objetivos más definidos:

- **Ingeniería inversa de la versión actual:** Análisis en profundidad el estado actual del dispositivo, entendiendo su funcionamiento e identificando posibles mejoras a implementar.

- **Almacenamiento de perfiles en el microcontrolador:** Utilización de la memoria EEPROM del microcontrolador para almacenar datos que convenga mantener de forma no volátil.
- **Funcionalidad compatible con faros de baja velocidad:** Implementación de un nuevo modo de funcionamiento que permita también probar faros de baja velocidad, es decir, que requieran señales de una frecuencia inferior a las que el dispositivo está programado para producir.
- **Interfaz gráfica:** Diseño de una aplicación para ordenador que permita gestionar la configuración del dispositivo tanto desde Windows como desde Linux. Esta ha de ser capaz de obtener las configuraciones almacenadas en el PWM Box, guardarlas de forma permanente en el disco, y posteriormente volver a cargarlas en el dispositivo.

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 algunos de las tareas que se van a llevar a cabo.
- **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 al diseño de la aplicación.
- **Tecnologías:** En este apartado se hará una comparativa de las diferentes tecnologías y herramientas que se han planteado usar en el transcurso del desarrollo, detallando brevemente las motivación de cada una de las elecciones tomadas.
- **Diseño y desarrollo:** Esta parte se centrará en el proceso de ideaación y posterior creación de la solución final, explicando paso a paso distintos aspectos de interés del desarrollo.

Capítulo 2

Estado del arte

Como se ha comentado en la Sección 1.1, este trabajo continúa el comenzado por el alumno Rubén Sanchez en cursos anteriores. Por ello, el primer paso a realizar al revisar el estado del arte es una ingeniería inversa de su desarrollo, con el fin de identificar tanto sus fortalezas como sus puntos débiles.

2.1. Ingeniería inversa

Para este proceso, se recibe por parte del tutor un archivo comprimido con el código existente del proyecto, así como el prototipo actual del dispositivo, mostrado anteriormente en la Sección 1.1.

```
▶ 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_8channels_0_1.hex PWM_BOX_V21_EEPROM_WRITE.bat
eeprom_8channels_m.hex PWM_BOX_V21_programROM.bat
eeprom_blanked.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 2.1: Contenido de *PWM_BOX.RAR*.

Con estos últimos, se procede a realizar una primera clasificación básica para facilitar su manejo. Según su contenido y utilidad, se pueden distinguir los distintos 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 AVRDUDE¹. AVRDUDE 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, funcionando incluso de *back-end* en otros entornos de desarrollo.
- **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.

Con los archivos algo más organizados 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 distintos grupos:

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 recoje 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 salvapantallas.
- 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.

¹<https://github.com/avrdudes/avrdude>

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. [3]

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.
- **Señales lentas:** No está aún implementado, pero sí encontramos referencias a él, así como las cabeceras de algunas funciones que usaría.

Para cada uno de los menús mencionados, se encuentran en este archivo las definiciones de 4 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 usabilidad 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. Al fin y al cabo, todo el software que se utilice ha de ser compatible con ambos.

Sin embargo, en este caso, se ha decidido usar Linux por preferencia personal del desarrollador. 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 gran ventaja, 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

Lenguaje para el desarrollo del 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. [1]

Estando esta parte del proyecto basada en la programación de un Arduino, la 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 la eficiencia 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.

Lenguaje para el desarrollo de la 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 en la Sección 1.3. Debido a ello, así como a experiencias previas mencionadas en la Sección 1.2, 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, Android y otros sistemas empotrados sin necesidad de modificar el código base. Qt proporciona licencias comerciales, pero también está disponible como 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 Plasma², un entorno gráfico *open-source* incluido en la mayor parte de las distribuciones más usadas.

Qt fue creada para ser usada con C++, pero en la actualidad tiene soporte para otros lenguajes de programación, como Python. Esta versión resulta interesante, ya que una característica a destacar de Python es su extensibilidad a través del uso de módulos, que podría facilitar la integración de la aplicación con el dispositivo.

²<https://kde.org/plasma-desktop/>

2.2.3. Entorno de desarrollo

Entorno para la implementación del 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 autocompletación de código.

Es por ello que finalmente he decidido usar un editor más genérico, como Visual Studio Code. Este es un editor *open-source* muy ligero, altamente 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 usaré 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 para la herramienta del mismo nombre a través de *snippets* de texto.

Entorno para la implementación de la interfaz En cuanto a la interfaz, la elección de software no es tan importante, puesto que el código en Python puede ejecutarse desde la propia *shell* sin problema. En un primer lugar me decanté por usar también Visual Studio Code, de forma que podría monitorizar la ejecución de ambos códigos al mismo tiempo. Sin embargo, tras usarlo unos días, me di cuenta que el *language server* de Python para VSCode, Jedi, se volvía extremadamente lento al incluir las distintas librerías de Qt. Esto hacía que no se procesara la sintaxis del lenguaje, y algunas características como la autocompletación de código o la detección de errores no funcionaran correctamente, lo cual hacía el trabajo algo más incómodo.

Es por ello que finalmente acabé instalando otro servidor, PyLSP, en

Neovim, un editor de texto muy ligero integrado en la terminal comúnmente usado en Linux. El problema no se solucionó del todo, ya que, según he encontrado en internet, parece ser que las librerías de Qt son bastante pesadas, y ralentizan considerablemente cualquier *language server* de Python, pero sí que observé alguna mejoría, suficiente para trabajar algo más cómodamente.

2.3. Planificación temporal

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 [...], 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. Se llevaría a cabo de la siguiente forma:

- 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 *front-end* sin completar primero las características del *back-end*.
- 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. Esto no resulta una desventaja, dado que los requisitos del proyecto están bien definidos desde el principio. En lo que a la metodología ágil respecta, se aprovecha la frecuente iteración, que asegura la calidad del producto final.

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

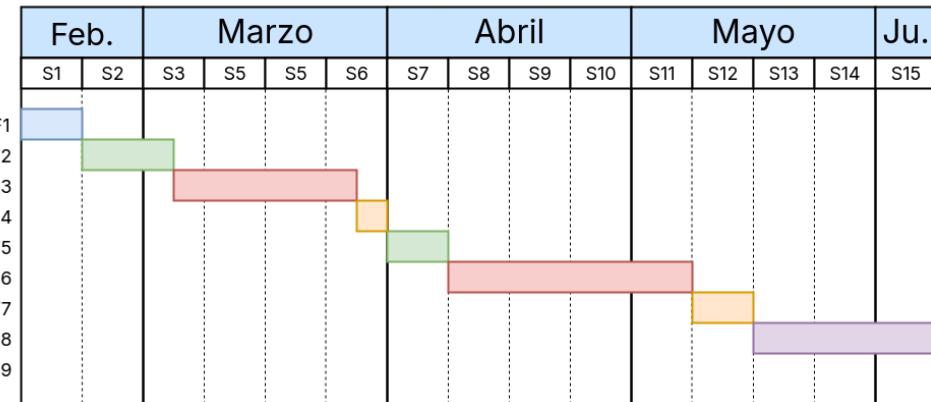


Figura 2.2: Diagrama de Gantt que muestra la organización de las distintas fases del proyecto a lo largo de las semanas.

- La fase de color azul representa un breve análisis inicial de los requisitos, para resolver las dudas que puedan surgir acerca de los objetivos del trabajo.
- Las verdes corresponden al tiempo de diseño. En el caso del firmware conviene ser algo más generoso, teniendo en cuenta el tiempo de adaptación inicial al proyecto.
- 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 un repaso final.
- 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.

Capítulo 3

Firmware

3.1. 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 1.1, 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

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 3.1: RF-1. Guardar 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 3.2: RF-2. Cargar 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

Cuadro 3.3: RF-3. Eliminar perfiles.

Gestión de la configuración del dispositivo

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 3.4: RF-4. Guardar la contraseña.

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

Cuadro 3.5: RF-5. Guardar el brillo de la pantalla.

Funcionalidad de señales lentes

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

Cuadro 3.6: RF-6. Mostrar secuencias disponibles.

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

Cuadro 3.7: RF-7. Ejecutar secuencias lentes.

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

Cuadro 3.8: RF-8. Parar la ejecución de secuencias lentas.

3.2. 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.

3.2.1. 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 3.2 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.

3.2.2. 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 prevee 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:

`^T,C(,P1,P2 ... ,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.

3.2.3. 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 compartmentalizará 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, descrito en la sección 3.1, no se estima necesario encriptar la contraseña de ninguna forma.

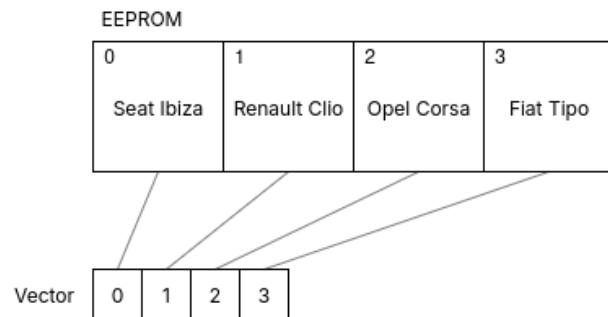
Dada la capacidad del usuario de añadir, borrar y reemplazar ranuras de memoria según desee, se prevee la necesidad de algún mecanismo adicional para determinar qué ranuras mostrar en el menú principal del dispositivo. Con este fin, se incluirá también un vector auxiliar, que contendrá el índice de las 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 3.1

3.2.4. General

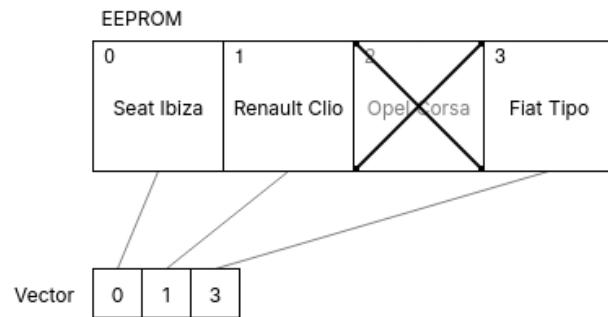
La organización general de los archivos dentro del proyecto también se considera un aspecto importante a tener en cuenta para facilitar su manejo. Tras los cambios planteados, algunos de los cuales incluyen la creación de aún más ficheros, esta necesidad se hace aún más aparente.

Para tratarlo, se plantea agruparlos en distintos directorios, a los que llamaremos módulos, de forma que queden distribuidos como se muestra en la figura 3.2.

- **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.



Usuario borra la ranura 2:



Usuario añade una ranura nueva:

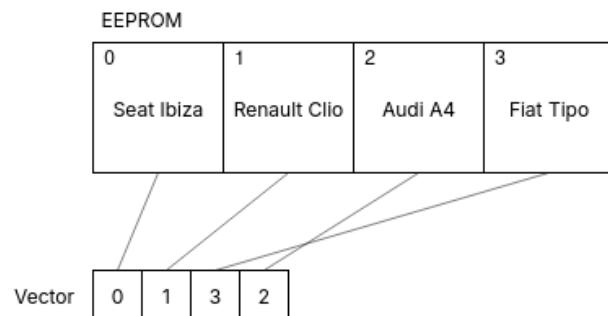


Figura 3.1: Representación del funcionamiento del vector auxiliar al realizar distintas operaciones sobre la EEPROM.

De esta forma, los distintos archivos quedan más agrupados según su fun-

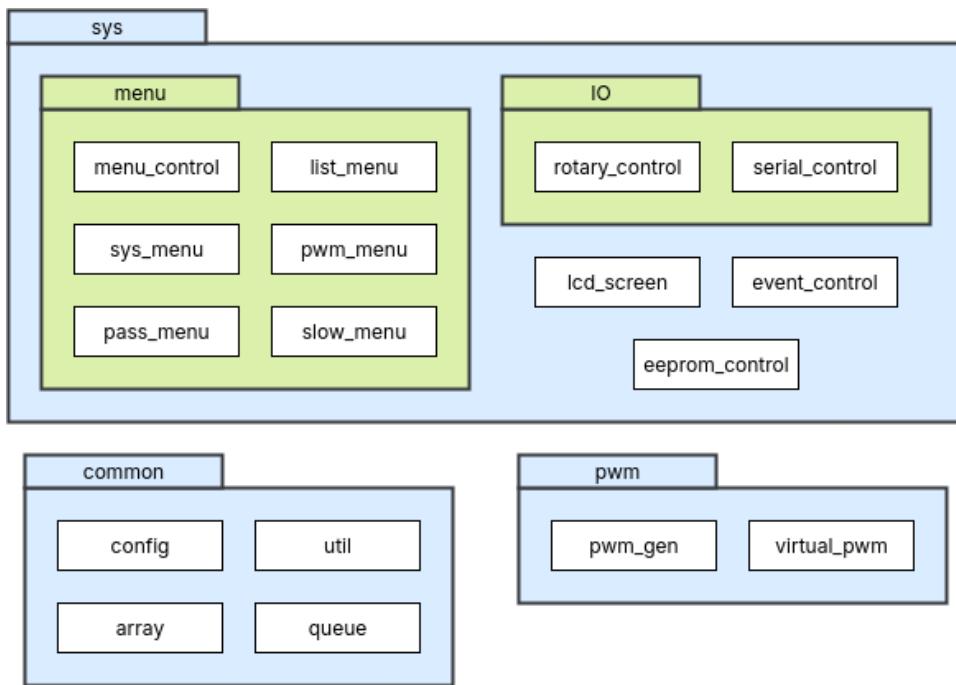


Figura 3.2: Distribución del código en distintos archivos y módulos.

cionalidad, 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 3.3. Adicionalmente, la figura 3.4 muestra el manejo de las interrupciones.

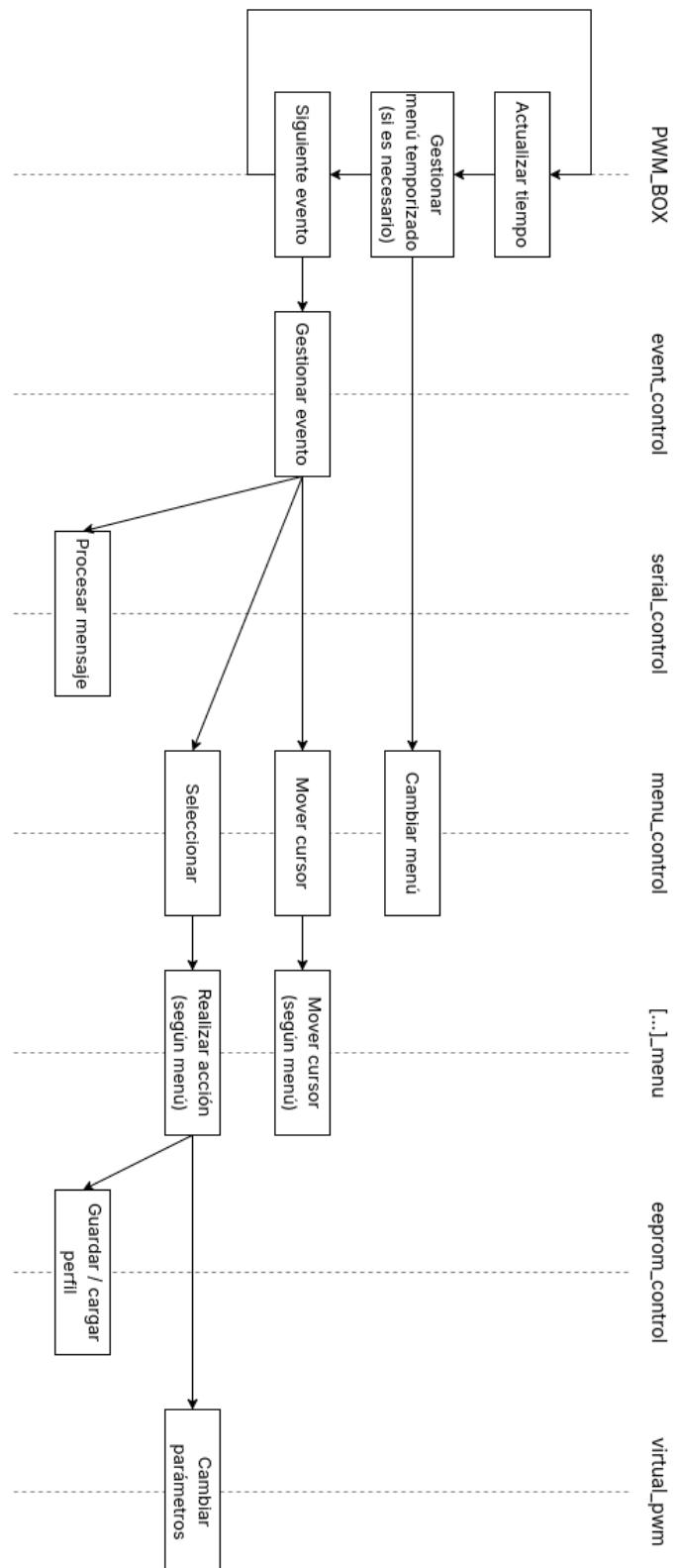


Figura 3.3: Ejecución del bucle principal.

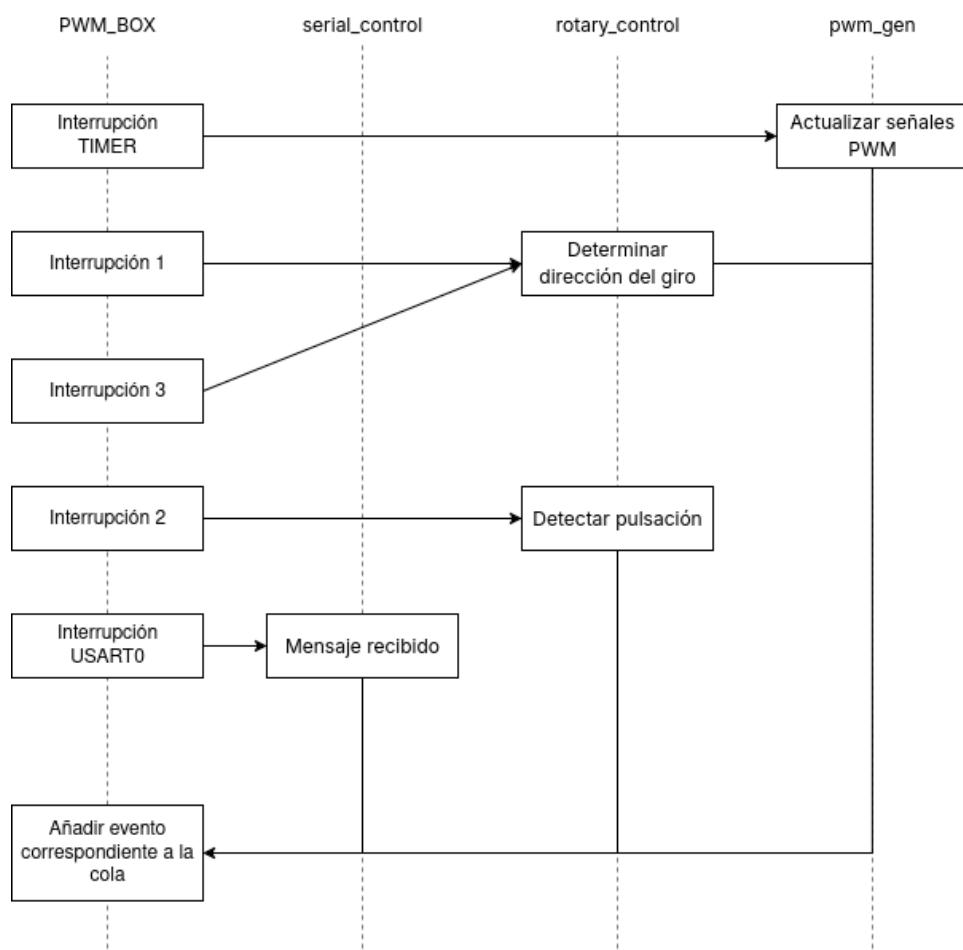


Figura 3.4: Ejecución de las distintas interrupciones.

3.3. Desarrollo

En esta sección se detallarán las decisiones enfrentadas durante la implementación del firmware, centrando 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[2], microcontrolador incluido en el Arduino Mega 2560 del que se hace uso en este proyecto.

3.3.1. Menú de señales lentas

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 subsección). Cuando esto ocurra, el sistema mostrará esta pantalla.

3.3.2. *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.

3.3.3. 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: [2]

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

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 3.2.

3.3.4. 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 3.2.

A estas se une el vector auxiliar también mencionado en la sección 3.2. 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 3.5.

```
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 3.5: Contenido de la memoria EEPROM.

3.4. Pruebas

Debido a la naturaleza del trabajo, la mayoría de pruebas se han realizado de manera experimental, observando los efectos de los cambios realizados sobre los distintos elementos.

Para comprobar 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. En la figura 3.6 se muestran las señales PWM resultantes de distintas configuraciones. Adicionalmente, en las figuras 3.7 y 3.8, se ven dos de las secuencias de señales lentas incluidas con el dispositivo.

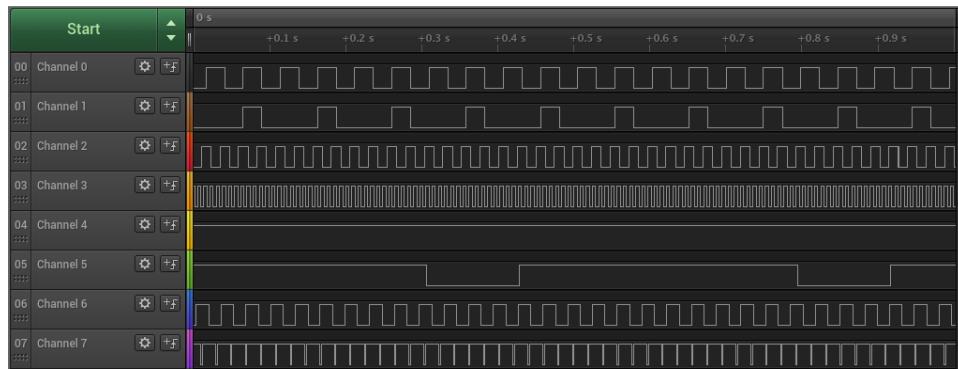


Figura 3.6: Señales PWM con distintas configuraciones.



Figura 3.7: Una de las secuencias de señales lentas preconfiguradas.

Estas pruebas han sido realizadas con el Saleae Logic v1.2.40¹, que es software propietario. Sin embargo, también podría usarse una versión open-source como puede ser Sigrok PulseView v0.4.2².

¹<https://support.saleae.com/logic-software/legacy-software/older-software-releases>. Accedido el 28 de agosto de 2025.

²<https://sigrok.org/wiki/Downloads>. Accedido el 28 de agosto de 2025.



Figura 3.8: Otra de las secuencias de señales lentas preconfiguradas.

Capítulo 4

Interfaz gráfica

4.1. 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 3.1. 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

ID	RF-1.1
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 4.1: RF-1.1. Conectar con el dispositivo.

ID	RF-1.2
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 4.2: RF-1.2. Obtener la información básica del dispositivo.

ID	RF-1.3
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 4.3: RF-1.3. Obtener la configuración general del dispositivo.

ID	RF-1.4
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 4.4: RF-1.4. Obtener las ranuras guardadas en el dispositivo.

ID	RF-1.5
Nombre	Enviar configuración general al dispositivo
Descripción	El sistema debe ser capaz de enviar de vuelta al dispositivo su configuración básica.
Prioridad	Alta

Cuadro 4.5: RF-1.5. Enviar configuración general al dispositivo.

ID	RF-1.6
Nombre	Enviar ranuras al dispositivo
Descripción	El sistema debe poder enviar ranuras configuradas al dispositivo.
Prioridad	Alta

Cuadro 4.6: RF-1.6. Enviar ranuras al dispositivo.

Gestión del dispositivo

ID	RF-2.1
Nombre	Mostrar la información básica del dispositivo
Descripción	El sistema debe permitir al usuario consultar la información básica del dispositivo.
Prioridad	Alta

Cuadro 4.7: RF-2.1. Mostrar la información básica del dispositivo.

ID	RF-2.2
Nombre	Mostrar la configuración general del dispositivo
Descripción	El sistema debe al usuario visualizar la configuración general del dispositivo.
Prioridad	Alta

Cuadro 4.8: RF-2.2. Mostrar la configuración general del dispositivo.

ID	RF-2.3
Nombre	Modificar la configuración general del dispositivo
Descripción	El sistema debe permitir al usuario establecer una nueva contraseña y definir cuál será la ranura cargada por defecto.
Prioridad	Alta

Cuadro 4.9: RF-2.3. Mostrar la configuración general del dispositivo.

Gestión de ranuras

ID	RF-3.1
Nombre	Crear ranuras nuevas
Descripción	El sistema debe permitir al usuario crear ranuras nuevas.
Prioridad	Alta

Cuadro 4.10: RF-3.1. Crear ranuras nuevas.

ID	RF-3.2
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 4.11: RF-3.2. Exportar ranuras al equipo.

ID	RF-3.3
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 4.12: RF-3.3. Importar ranuras del equipo.

ID	RF-3.4
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 4.13: RF-3.4. Visualizar la configuración de las ranuras.

ID	RF-3.5
Nombre	Modificar los parámetros de las ranuras cargadas
Descripción	El sistema debe permitir la creación de ranuras nuevas desde la propia aplicación.
Prioridad	Alta

Cuadro 4.14: RF-3.5. Obtener las ranuras guardadas en el dispositivo.

4.2. Diseño

La fase de diseño de la interfaz se comienza con un repaso de los requisitos. Teniendo en cuenta que su objetivo principal es ser usada en la industria automotriz, desde el comienzo del diseño se planteó un producto simple, con la mayor cantidad de información posible a simple vista.

Esta idea inicial fue variando ligeramente conforme fuimos concretando algunas de las características a implementar, pero siempre se mantuvo fiel a su diseño original.

El resultado final

4.2.1. JSON Manager

4.2.2. PWM Types

4.2.3. PWM Box

4.2.4. Main Window

4.3. Desarrollo

4.4. Pruebas

Bibliografía

- [1] B. Dennis M. Ritchie: *The C Programming Language*. Prentice-Hall software series. Prentice Hall, 2nd ed., 1988, ISBN 9789688802052.
- [2] 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.
- [3] 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.

