



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



**TFG del Grado en Ingeniería
Informática**

CRET
**CanBus Reverse Engineering
Toolkit**



Presentado por Adrián Marcos Batlle
en Universidad de Burgos — 13 de febrero
de 2019

Tutores: Álvar Arnaiz González y César
Repsa Pérez



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. Álvar Arnaiz González profesor del departamento de Ingeniería Civil, área de Lenguajes y Sistemas Informáticos, y D. César Represa Pérez, profesor del departamento de Ingeniería Electromecánica.

Exponen:

Que el alumno D. Adrián Marcos Batlle, con DNI 71310384B, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado CRET - CanBus Reverse Engineering Toolkit.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 13 de febrero de 2019

Vº. Bº. del Tutor:

D. Álvar Arnaiz González

Vº. Bº. del co-tutor:

D. César Represa Pérez

Resumen

El bus CAN (*CAN Bus - Controlled Area Network*) es un protocolo de comunicación utilizado en los desarrollos de multitud de sectores críticos como la industria, la automoción y la aviación entre otros, para la comunicación entre los componentes internos que forman la infraestructura desarrollada.

Los datos que son transmitidos por dicho bus son propiedad de cada uno de los fabricantes, a pesar de ser un protocolo con los estándares públicos.

El desarrollo de este proyecto viene motivado a realizar un análisis de la información que circula por estos buses, así como su clasificación y monitorización en tiempo real.

Para dicho objetivo, se ha desarrollado tanto una parte de *software* (para el análisis, clasificación y monitorización de los datos), como una parte de *hardware* necesaria para conectarse a dicho bus.

Con ello se persigue el objetivo de, por ejemplo, en caso de que necesitemos obtener una señal de un sensor conectado a dicha red, en lugar de instalar otro sensor nuevo, sería posible obtener el valor del ya instalado una vez haya sido identificado.

Descriptores

Bus CAN, Automoción, Industria, Aviónica, Ingeniería inversa, Análisis de protocolos.

Abstract

CAN Bus (*Controlled Area Network*) is a communication protocol used in the development of a large number of critical sectors such as industry, automotive and aviation among others, for communication between internal electronic components.

Data transmitted through the CAN bus is property of each developer, despite being a protocol which standards are public.

The development of this project is motivated to perform an analysis of the data that flows through these buses, as well as their classification and real time monitoring.

For this purpose, both custom software and hardware have been developed to achieve the requirements.

This means we can, for example, getting the signal of a sensor connected to the bus, instead of installing a new one.

Keywords

Can Bus, Automotive, Industry, Avionics, Reversing, Protocol analysis.

Índice general

Índice general	III
Índice de figuras	v
Índice de tablas	vi
Introducción	1
1.1. Estructura de la memoria	2
1.2. Materiales adjuntos	2
Objetivos del proyecto	5
2.1. Objetivos generales	5
2.2. Objetivos técnicos	5
2.3. Objetivos personales	6
Conceptos teóricos	7
3.1. Bus CAN	7
3.2. Desarrollo del hardware	12
3.3. Desarrollo del software	15
Técnicas y herramientas	19
4.1. Metodologías	19
4.2. Patrones de diseño	19
4.3. Control de versiones	21
4.4. Gestión del proyecto	22
4.5. Entorno de desarrollo	22
4.6. Documentación	23

4.7. Impresión 3D	23
4.8. Librerías	23
4.9. Diseño de hardware	24
4.10. Bus CAN	25
Aspectos relevantes del desarrollo del proyecto	27
5.1. Inicio del proyecto	27
5.2. Metodologías	28
5.3. Formación	28
5.4. Desarrollo del <i>hardware</i>	28
5.5. Desarrollo del <i>software</i>	33
5.6. Diseño de la interfaz gráfica	35
5.7. Testing	35
5.8. Diseño e impresión 3D de una caja protectora	36
Trabajos relacionados	39
Conclusiones y Líneas de trabajo futuras	41
7.1. Conclusiones	41
7.2. Líneas de trabajo futuras	42
Bibliografía	43

Índice de figuras

3.1. Ejemplo de dispositivos conectados a un bus CAN.	7
3.2. Esquema de conexión de los nodos en el bus CAN.	8
3.3. Señales eléctricas del bus CAN vistas por un osciloscopio.[6]	9
3.4. Anatomía de un nodo CAN.	9
3.5. Identificación de los campos que componen un <i>Frame</i>	10
3.6. Ejemplo de varios componentes a nivel esquemático.	12
3.7. Esquema de pistas de una PCB.	13
3.8. Ilustración con 4 <i>pads</i> en una PCB.	13
3.9. Visión de una vía a través de un corte de una PCB.	14
3.10. Ejemplo de una huella de un IC (<i>Integrated Circuit</i>).	14
3.11. Ejemplo del tipo <i>LineChart</i> en JavaFX.	16
3.12. Etapas del arranque de una aplicación JavaFX.[15]	17
3.13. Ejemplo de <i>ToolTip</i> en JavaFX.	17
4.14. Modelo MVC.	20
4.15. Representación de la arquitectura de la aplicación.	21
5.16. Vista general del esquema.	30
5.17. Esquema de la PCB del proyecto.	31
5.18. PCB del proyecto sin los componentes.	32
5.19. PCB del proyecto con los componentes soldados.	32
5.20. Pines conectados al microcontrolador PIC para la carga del <i>firmware</i>	33
5.21. Placa casera para simular una red CAN.	36
5.22. Renderizado de la caja en <i>Blender</i>	36
5.23. Caja impresa en 3D cerrada.	37
5.24. Caja impresa en 3D abierta.	37

Índice de tablas

6.1. Comparación entre CRET y otras herramientas del mercado . . . 40

Introducción

Todos los vehículos que utilizamos en el día a día, maquinaria utilizada en las empresas, el sector náutico o la aviación, utilizan el bus CAN para la *interconexión* de los componentes electrónicos que hacen funcionar dichas máquinas.

Los estándares del protocolo son públicos de forma que cualquier desarrollador pueda ceñirse a ellos, pero no sucede así con los datos que circulan por el bus, ni la forma en la que lo hacen.

La forma en la que se envían y reciben esos datos, al igual que el propio dato en si, es definido por cada fabricante, y no se suelen hacer públicos.

Lo que se intenta conseguir a través del uso de esta herramienta, es la posibilidad identificar, clasificar y monitorizar los datos que los distintos elementos de la máquina (un vehículo, por ejemplo) utilizan para su funcionamiento interno y para el intercambio tanto de datos como de señales o textos.

De esta manera, por ejemplo, si necesitásemos realizar una aplicación para la monitorización de un vehículo y ciertas señales como la velocidad, las revoluciones del motor o el GPS fueran transmitidas a través del bus CAN, no sería necesario introducir nuevos sensores, sino que estos datos serían extraídos del bus CAN, ahorrando costes, facilitando el desarrollo y evitando posibles problemas.

Para realizar ese análisis existen algunos programas y proyectos disponibles, pero que requieren de un *hardware* con un precio elevado. Además, es necesario una gran cantidad de conocimientos previos para el uso de dichas herramientas.

Con este proyecto, se propone realizar dicha clasificación y monitoriza-

ción de manera rápida e interactiva, representando los datos visualmente, permitiendo etiquetarlos y almacenarlos de manera que puedan ser utilizados posteriormente.

1.1. Estructura de la memoria

A continuación se describe la estructura de la memoria:

- **Introducción:** Descripción breve sobre el proyecto, motivación por la que se ha realizado y soluciones propuestas. Estructura de la memoria y listado de materiales adjuntos proporcionados.
- **Objetivos del proyecto:** Exposición de los objetivos, clasificados en objetivos generales, objetivos técnicos y objetivos personales.
- **Conceptos teóricos:** Conceptos básicos y necesarios para entender el propósito del proyecto así como su desarrollo.
- **Técnicas y herramientas:** Metodologías y herramientas utilizadas durante el desarrollo del proyecto.
- **Trabajos relacionados:** Aplicaciones, proyectos y empresas que ofrecen soluciones en el mismo campo que el estudiado.
- **Conclusiones y líneas de trabajo futuras:** Conclusiones a las que se ha llegado tras la realización del proyecto, así como mejoras y futuro desarrollo de la aplicación.

1.2. Materiales adjuntos

Los materiales adjuntos a la memoria son los siguientes:

- Aplicación desarrollada en Java: CRET.
- Fotos del *hardware* desarrollado.
- Esquemas del *hardware* desarrollado.
- Diseño de la caja preparada para la impresión 3D.
- Documentación del código fuente, en formato JavaDoc.
- Vídeos de prueba del proyecto.

Además, los siguientes recursos están accesibles a través de internet:

- Repositorio del proyecto: <https://github.com/amb0070/CRET/>

Objetivos del proyecto

A continuación se definen los objetivos del proyecto realizado, estructurados en tres secciones:

2.1. Objetivos generales

- Desarrollar una aplicación para el análisis y monitorización de los datos que fluyen por el bus CAN.
- Desarrollo de un *hardware* libre el cual permita conectarse a dicho bus de datos y monitorizar más de un bus de forma simultánea.

2.2. Objetivos técnicos

- Diseñar y desarrollar un *hardware* propio desde 0 siguiendo la metodología para el diseño del mismo, así como su producción y montaje.
- Desarrollar una aplicación en Java con el uso de la librería JavaFX para la parte correspondiente a la interfaz gráfica.
- Aplicar la arquitectura MVC (*Model-View-Controller*) en el desarrollo de la aplicación.
- Uso de estructuras de datos que permitan su modificación de forma concurrente.
- Utilizar librerías para la recolección de datos del bus CAN.

2.3. Objetivos personales

- Profundizar en el conocimiento de *hardware* y en el desarrollo del mismo.
- Adquirir conocimiento sobre el funcionamiento del bus CAN en distintos escenarios.
- Profundizar en el conocimiento del análisis de datos y monitorización en tiempo real.

Conceptos teóricos

Las partes del proyecto con mayor desconocimiento y complejidad están enfocadas principalmente en el funcionamiento del bus CAN y en el desarrollo del *hardware*, el cual requiere de unos conocimientos básicos y unas metodologías específicas las cuales serán detalladas a continuación:

3.1. Bus CAN

El bus CAN (*Controller Area Network*)^[19] es un protocolo desarrollado para la comunicación entre los distintos micro-controladores y dispositivos que son necesarios para el funcionamiento de una máquina (un vehículo, por ejemplo). Este protocolo no necesita de un host principal, sino que sigue una topología de tipo "bus".

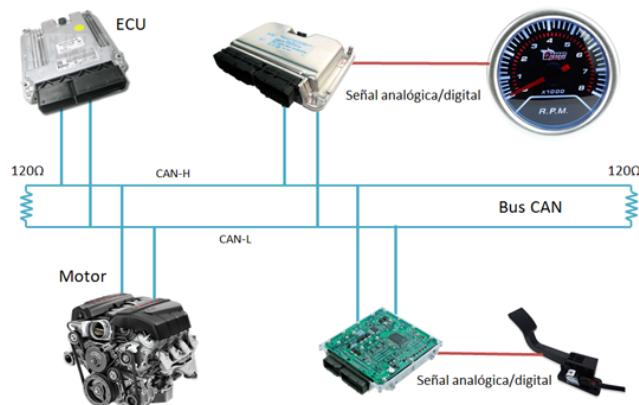


Figura 3.1: Ejemplo de dispositivos conectados a un bus CAN.

Dicho protocolo está basado en el uso de mensajes para el intercambio de información entre los distintos dispositivos que lo componen.

Es utilizado en multitud de escenarios como la aviación, la navegación, la automatización industrial, instrumentos médicos, maquinaria pesada y ascensores, entre otros.

Una de las características que nos interesa conocer, es que al ser una topología de tipo BUS, todos los nodos tienen acceso a la información que es transmitida a través del bus, con lo que es suficiente conectarlos a dicho bus y capturar los datos que viajan a través de él para analizarlos posteriormente.

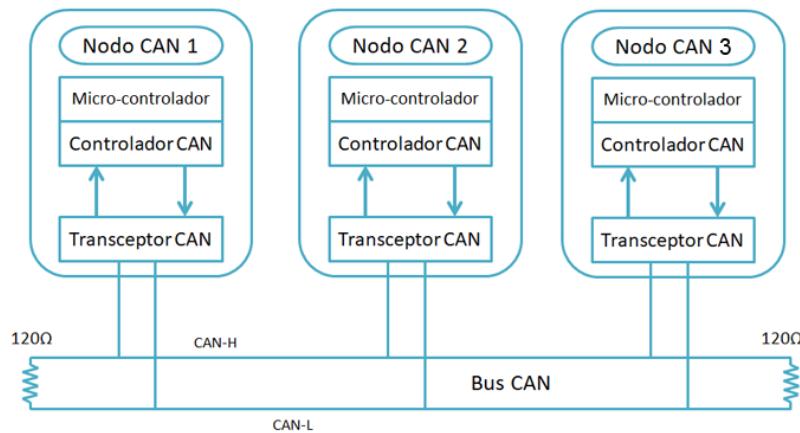


Figura 3.2: Esquema de conexión de los nodos en el bus CAN.

El principal elemento que compone una red de tipo CAN son los nodos, además de los cables de conexión entre los mismos:

- **Nodo:** Cada uno de los dispositivos físicos que están conectados a la red CAN. Al menos es necesario que existan dos nodos conectados a la red para que se produzca una comunicación.
- **Cable:** Se trata del cable a través del cual se envían y reciben los datos. Todos los nodos deben de estar conectados a estos dos cables. Estos son identificados por CAN-H (*CAN-High*) y CAN-L (*CAN-Low*).

Esto es debido a que las señales que circulan por los mismos, son señales diferenciales, de las cuales el valor efectivo interpretado por el nodo es extraído tras aplicar la diferencia entre ambas como se puede ver en la figura.

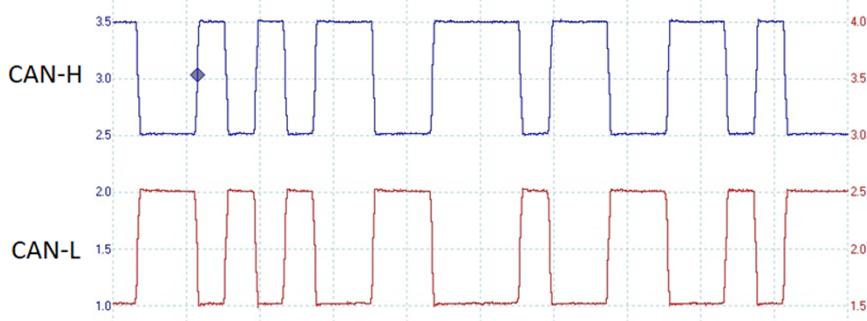


Figura 3.3: Señales eléctricas del bus CAN vistas por un osciloscopio.[6]

Cada uno de los nodos es capaz de enviar y recibir mensajes. La prioridad de los mensajes de un nodo viene dada por la ID(identificador) del *Frame* enviado al bus.

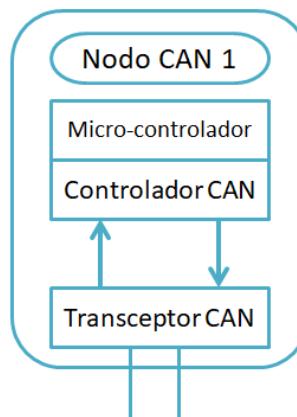


Figura 3.4: Anatomía de un nodo CAN.

A continuación se describen los elementos de una trama CAN, así como la descripción de la misma.

Frame (Trama): Cada uno de los mensajes enviados a través de la red CAN.

La estructura y el funcionamiento interno es más complejo, pero no nos centraremos en ellos ya que no tiene relevancia en este caso. Únicamente nos centraremos en los tres elementos identificados en la figura.

Cada uno de los *Frames* contiene tres campos en los que nos centraremos: ID, Length y Data.

- **ID (Identificador):** Se trata del identificador del *Frame*. A través de él, se establece la prioridad que tiene ese *Frame* durante el acceso al bus de datos.
- **Length (Longitud):** En este campo, se define la longitud del campo de datos del *Frame*. En las versiones estándar, esta longitud puede ir desde 0 hasta 8 *bytes* de datos.
- **Data (Datos):** Campo en el que se transportan los datos que son enviados por el nodo correspondiente. La longitud de este campo puede ir desde 0 *bytes* hasta un total de 8 *bytes*.

ID	Longitud	Datos
095	[8]	80 00 07 F4 00 00 00 17
1A4	[8]	00 00 00 08 00 00 00 10
1AA	[8]	7F FF 00 00 00 00 67 11
1B0	[7]	00 0F 00 00 00 01 57
1D0	[8]	00 00 00 00 00 00 00 0A
166	[4]	D0 32 00 27
158	[8]	00 00 00 00 00 00 00 28
161	[8]	00 00 05 50 01 08 00 2B
191	[7]	01 00 10 A1 41 00 1A
133	[5]	00 00 00 00 B6

Figura 3.5: Identificación de los campos que componen un *Frame*.

Bitrate: Se trata de la velocidad a la que los datos son transmitidos a través del bus. Todos los nodos deben de transmitir a la misma velocidad para entenderse entre ellos. Esta, puede cambiar dependiendo de los sistemas y de su desarrollo. Los valores más comunes son:

- 10 000 bit/s
- 20 000 bit/s
- 50 000 bit/s
- 100 000 bit/s
- 125 000 bit/s
- 250 000 bit/s
- 500 000 bit/s
- 800 000 bit/s
- 1 000 000 bit/s

Además, el estándar dispone de varias medidas para la detección de errores y seguridad las cuales no son relevantes para la exposición de este proyecto.

El desarrollo del *hardware* con dos interfaces viene motivado a la posibilidad de que exista más de un bus en una misma máquina. De esta manera, sería posible el análisis de dos buses de forma simultánea.

Siguiendo el modelo OSI[2], el estándar bus CAN especifica únicamente las dos primeras capas:

- **Capa física:** Esta capa define cómo son transmitidas las señales a nivel eléctrico, es decir, los niveles de las señales en su representación como bits, y el medio de transmisión que va a ser utilizado. Además, se encargaría de gestionar la sincronización de los mensajes, así como su *bit encoding*.
- **Capa de enlace de datos:** Esta capa se encarga de la parte lógica del protocolo. Entre sus funciones se encuentra el filtrado de los *Frames*, detección de sobrecarga del bus y la detección de errores.

Existen 3 modos por los cuales los nodos pueden conectarse al bus CAN:

- **Active:** Cuando un nodo es conectado en modo *active*, significa que si envía un mensaje a través del bus. Al no ser que el *transceiver* esté conectado en modo *Single Shot*, el mensaje será enviado continuamente por el bus hasta que se reciba un ACK (*Acknowledgment*). De manera que este modo si que tiene influencias sobre el bus.
- **Listen-Only:** Este modo no interfiere en el funcionamiento del bus CAN. Cuando un nodo es conectado en modo *listenonly*, dicho nodo no tiene influencia sobre el resto. Si por ejemplo llega un *Frame* erróneo, no genera *flags* de error. Se podría decir que es un modo de escucha del bus, sin posibilidad de interacción con el mismo.
- **LoopBack:** Se podría considerar una especie de modo *debug*. Cuando un nodo se conecta en modo *LoopBack*, significa que este nodo es capaz comunicarse consigo mismo, sin enviar ni recibir mensajes del bus. Es un modo utilizado solo para la realización de pruebas, no para aplicaciones finales.

3.2. Desarrollo del hardware

A continuación se describen distintos conceptos básicos sobre el *hardware* desarrollado y algunos de sus componentes:

Esquema

Representación esquemática de los componentes de un circuito, así como sus entradas y salidas, y conexión entre ellos.

A continuación se muestra un ejemplo visual de varios componentes en un esquema para el diseño de una PCB(*Printed Circuit Board*):

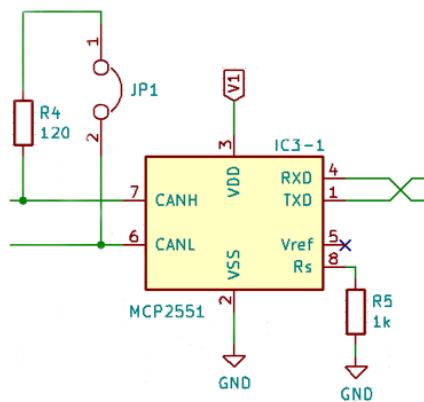


Figura 3.6: Ejemplo de varios componentes a nivel esquemático.

PCB - Printed Circuit Board

Hace referencia a la placa sobre la que van soldados todos los componentes eléctricos necesarios para hacer funcionar el circuito. Suele tener 3 componentes principales:

- Pistas
- *Pads*
- Vías

Pistas

Cada una de las conexiones entre los componentes. Hacen el trabajo de un cable, solo que integrado en la placa.

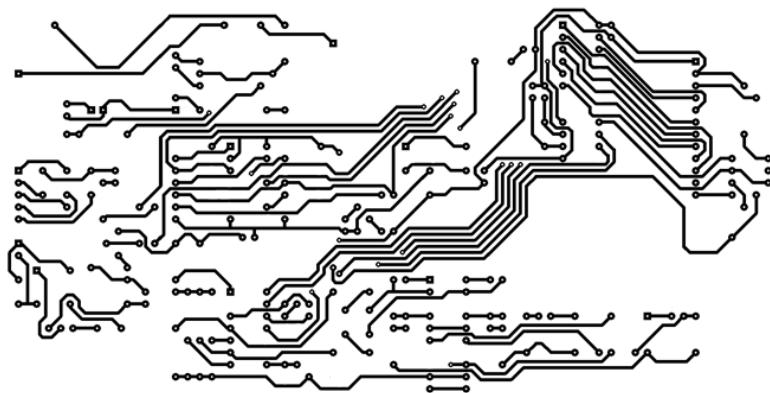


Figura 3.7: Esquema de pistas de una PCB.

Pad

Cada uno de los espacios en los cuales se sueldan los distintos elementos que componen el circuito.



Figura 3.8: Ilustración con 4 *pads* en una PCB.

Vía

Al tratarse de una PCB de doble capa (es decir, que tiene pistas o elementos en ambos lados), a veces es necesario realizar un cruce entre vías, o conectar un componente que está en el otro lado de la placa. Para ello se utilizan las vías, a través de ellas se consiguen conectar ambas capas de la PCB.



Figura 3.9: Visión de una vía a través de un corte de una PCB.

Footprint

Hace referencia al espacio necesario para colocar un elemento del *hardware*. Existen estándares que definen los tamaños y formas de dichos espacios.

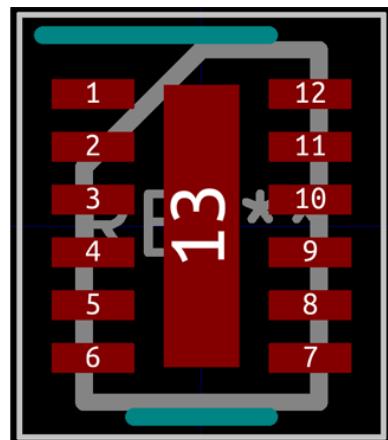


Figura 3.10: Ejemplo de una huella de un IC (*Integrated Circuit*).

Gerber

Es un formato de fichero el cual contiene la información necesaria para realizar la fabricación de una PCB.

Estos ficheros son aceptados por todos los fabricantes de placas de circuito impreso. Además es soportado por la mayoría de programas utilizados para el diseño de las mismas.

3.3. Desarrollo del software

Para el desarrollo del *software* se ha utilizado el lenguaje de programación Java. Además, para la mejora de la interacción con el usuario también se ha utilizado el módulo JavaFX y la librería *medusa*, un complemento para JavaFX.

Existen distintos elementos utilizados durante el desarrollo de la aplicación los cuales es interesante destacar:

- **GridPane:** Se trata de una estructura implementada en JavaFX la cual nos permite una distribución de los elementos que forman la interfaz gráfica en forma de cuadrícula.

En cada una de las celdas de esta cuadrícula se incluyen distintos elementos que se mencionarán a continuación.

- **LineChart:** Se trata de un módulo perteneciente a JavaFX el cual nos permite generar una gráfica. Es utilizado para mostrar los datos al usuario y que estos puedan ser identificados.

LineChart recoge los datos de una estructura de tipo *ObservableList*, en la cual se encuentran todos los datos a representar.

Este tipo de datos no permite el acceso concurrente de varios *threads* al mismo tiempo.

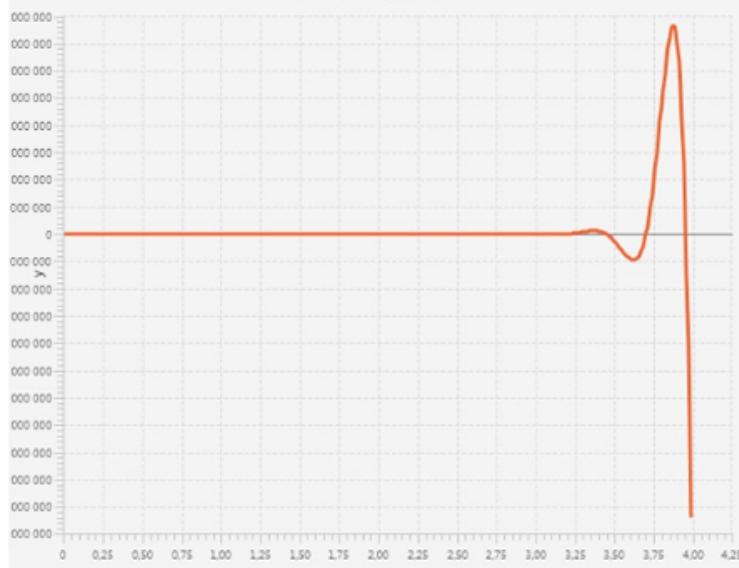


Figura 3.11: Ejemplo del tipo *LineChart* en JavaFX.

- **ConcurrentLinkedQueue:** A través del uso de esta estructura ha sido posible recoger los datos del bus CAN con un *thread* a la vez que el *thread* de la interfaz gráfica se encargaba de insertarlos en los objetos de tipo *ObservableList*.
- **FXML Document:** Se trata de documentos con un formato que extiende de XML. En estos documentos se encuentra la estructura de las interfaces gráficas y la distribución de los elementos en la misma. Este tipo de documentos pueden ser escritos a mano o generados con la ayuda de la herramienta *SceneBuilder*.
A este documento se le indica el *Controller* que va a manejarlo, de forma que se crea la conexión *View-Controller*. Esto se hace a través del campo *fx:controller* del documento FXML.
- **Preloader:** Debido a la cantidad de tiempo requerido para el arranque de la aplicación (comprobación de la base de datos, preparación de la interfaz gráfica), se decidió introducir un *preloader*. Este se encarga de mostrarnos una pantalla de carga mientras la aplicación realiza las tareas necesarias. En el momento en el que la pantalla principal está lista para mostrarse, esta manda una señal al *preloader* para que se cierre.

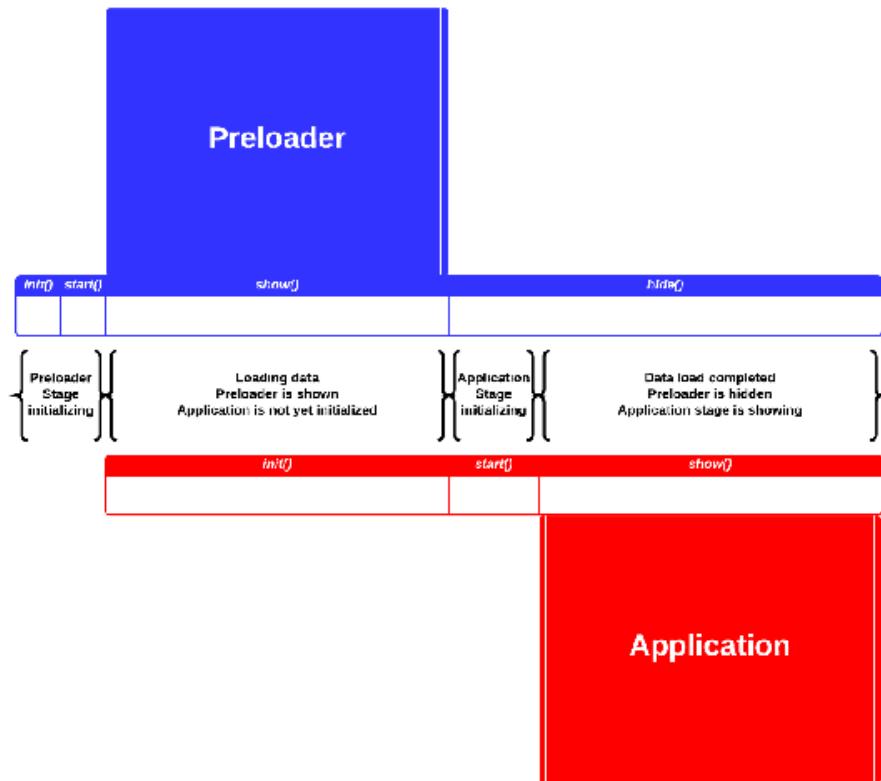


Figura 3.12: Etapas del arranque de una aplicación JavaFX.[\[15\]](#)

- **ToolTips:** Se trata de una descripción emergente de un elemento. Esta ayuda al usuario a entender la funcionalidad de un elemento simplemente arrastrando el cursor sobre el elemento.

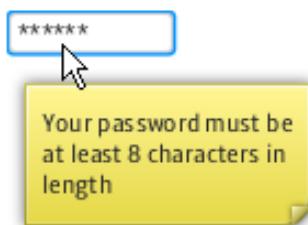


Figura 3.13: Ejemplo de ToolTip en JavaFX.

Técnicas y herramientas

4.1. Metodologías

Scrum

A través del uso de esta metodología, se adopta una estrategia de desarrollo incremental, en lugar de realizar una planificación y ejecución completa del proyecto desde el principio. Esto se consigue con el uso de *sprints*.

Pomodoro

El uso de este método de gestión de tiempo ayuda a incrementar la productividad. Esta técnica consiste en realizar una división del tiempo en fragmentos. Estos fragmentos son períodos de tiempo de 25 minutos (llamados *pomodoros*). A su vez, estos fragmentos están separados entre ellos por pausas de 5 minutos. Cuando se han realizado 4 períodos de 25 minutos (es decir, 4 *pomodoros*), se realiza una pausa más larga de unos 30 minutos.

Para realizar el seguimiento de esta técnica, se ha utilizado el siguiente recurso web: <https://tomato-timer.com/>

4.2. Patrones de diseño

MVC (*Model-View-Controller*)

Aplicando este patrón de arquitectura, se utilizan 3 componentes, las vistas, los modelos y los controladores, de tal forma que se separa la lógica de la vista de la aplicación. De esta forma, si realizamos una modificación en una parte de nuestro código, la otra parte no se ve afectada. Por ejemplo,

si modificamos la base de datos, solo modificaríamos el modelo encargado de esa acción, sin tocar el resto de la aplicación.

Está compuesto por tres componentes:

- **Model:** Normalmente esta parte se encarga de los datos (no siempre). Esto puede ser por ejemplo, consultando a una base de datos o cualquier otra estructura de datos utilizada en el desarrollo.
- **View:** Componen la representación visual de los datos, es decir, todo lo que tenga que ver con la interfaz gráfica.
- **Controller:** Es un mediador entre los modelos y las vistas. Se encarga de recibir las órdenes del usuario a través de la vista, realizar la petición de datos al modelo, y devolverlo de nuevo a la vista.

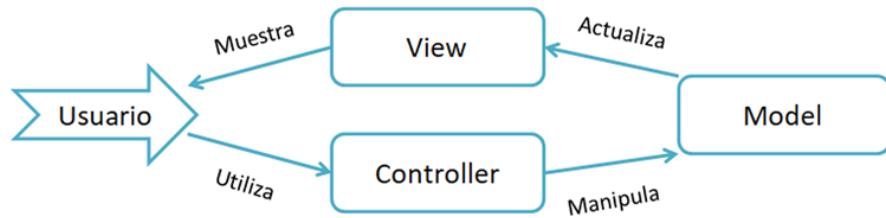


Figura 4.14: Modelo MVC.

Arquitectura de la aplicación

A continuación se muestra un esquema de la arquitectura simplificada de la aplicación.

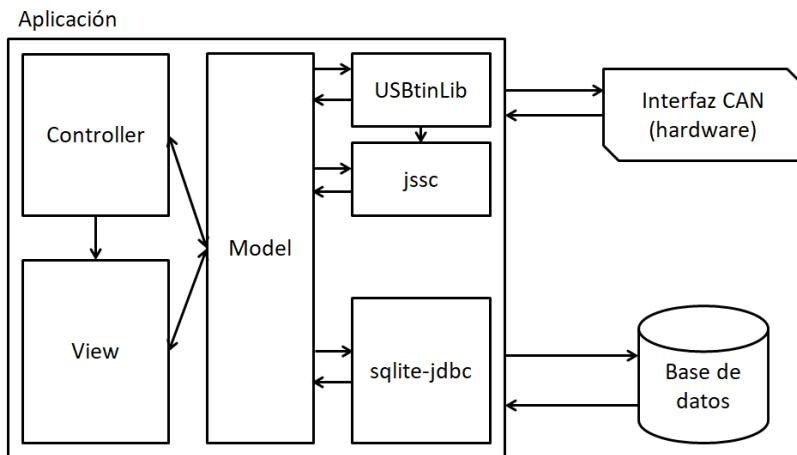


Figura 4.15: Representación de la arquitectura de la aplicación.

A grandes rasgos podemos observar la interacción con la base de datos a través de la librería *sqlite-jdbc*. Del mismo modo, se interacciona con la interfaz CAN a través del uso de la librería *USBtinLib*.

La librería *jssc* es la encargada de encontrar los puertos serie disponibles en el equipo.

Todo lo mencionado anteriormente es manejado por el modelo de la aplicación el cual interactúa con las el controlador y las vistas.

4.3. Control de versiones

Git

Con el uso de este sistema de control de versiones, se realiza de manera eficiente y confiable el mantenimiento de versiones de aplicaciones o proyectos, cuando estas poseen una gran cantidad de código fuente. Además, guarda un registro de los cambios realizados en la aplicación o proyecto, de forma que podemos volver a una versión anterior en cualquier momento.

GitHub

Se trata de una plataforma de desarrollo colaborativo en la cual es posible alojar proyectos de forma gratuita a través Git (el sistema de control de versiones utilizado por GitHub). Con el uso de una cuenta de estudiante, es posible crear un repositorio privado en esta plataforma.

4.4. Gestión del proyecto

ZenHub

Es una herramienta web integrada con *GitHub* desde la cual se puede realizar una gestión de proyectos. Para ello dispone de un tablero con tarjetas, las cuales son tareas del proyecto.

Estas tareas pueden estar en diversos estados a lo largo del proceso, y se les puede asignar prioridad en función de la posición en la lista.

Además, estas tareas pueden ser asignadas a un *sprint* concreto, y establecer un tiempo estimado para realizarlas.

Es gratuita para pequeños proyectos, y se puede encontrar en el siguiente enlace: <https://www.zenhub.com/>

4.5. Entorno de desarrollo

Eclipse IDE

Es un IDE (*Integrated Development Environment*) para el desarrollo de aplicaciones Java. Dispone de un gran número de *plugins* para facilitar el desarrollo, refactorización y revisión del código fuente.

Una de las decisiones del uso de esta IDE es la integración fácil y rápida que tiene con JavaFX, además de estar acostumbrado a utilizarla durante toda la carrera.

SceneBuilder

Es una interfaz de usuario con la cual a través del método *Drag and Drop*, es posible maquetar interfaces gráficas las cuales son utilizadas por JavaFX. Esta información es almacenada en ficheros FXML, un formato extendido de XML.

SQLite

Es un sistema de gestión de bases de datos de tamaño reducido, la cual no necesita de un servidor para ser utilizada, sino que los datos son almacenados en un único fichero en el sistema *host*.

4.6. Documentación

TeXMaker

Editor de L^AT_EX multiplataforma, el cual integra diversas herramientas necesarias para la generación de documentos L^AT_EX.

4.7. Impresión 3D

Blender

Software de tipo CAD (Computer Aided Design) el cual es utilizado para modelar piezas en 3D. En este caso ha sido utilizado para el desarrollo de la caja que contiene el *hardware* utilizado para conectarse al bus CAN.

Ultimaker Cura

Se trata de un *slicer*[18] gratuito para la impresión 3D. Es el encargado de transformar el diseño 3D en un formato que la impresora 3D sea capaz de procesar, en este caso un fichero *gcode*.

Página oficial:

<https://ultimaker.com/en/products/ultimaker-cura-software>

4.8. Librerías

USBtinLib

Librería utilizada para mediar entre el *hardware* utilizado para conectarse al bus CAN y Java. Es la encargada de realizar la conexión, además de que nos permite tanto enviar como recibir datos del bus.

Puede encontrarse más información sobre esta librería en el siguiente enlace: <https://github.com/EmbedME/USBtinLib>

SQLite-JDBC

Librería para realizar conexiones con bases de datos SQLite.

Puede encontrarse más información en el siguiente enlace:

<https://bitbucket.org/xerial/sqlite-jdbc>

JSSC (*Java Simple Serial Connector*)

Librería multiplataforma a través de la cual es posible obtener los puertos serie de comunicación disponibles en el equipo en el que se ejecute.

Puede encontrarse información en el siguiente enlace:

<https://github.com/scream3r/java-simple-serial-connector>

JavaFX

Se trata de una librería para Java, enfocada a la creación de interfaces gráficas. Dispone de multitud elementos predefinidos los cuales nos permiten la creación de una interfaz de uso fácil e intuitivo, además de ser agradable a la vista.

Puede encontrarse información en el siguiente enlace:

<https://docs.oracle.com/javase/8/javase-clienttechnologies.htm>

Medusa

Conjunto de componentes los cuales pueden ser añadidos a *JavaFX* para ampliar sus funcionalidades y características visuales.

Puede encontrarse en el siguiente enlace:

<https://community.oracle.com/docs/DOC-992746>

4.9. Diseño de hardware

KiCAD

Está compuesto por un conjunto de herramientas para el diseño de circuitos electrónicos. Esto incluye la creación de esquemas para circuitos electrónicos así como la conversión de esos esquemas a diseños de PCB (Printed Circuit Board). Además nos permite exportar los ficheros necesarios para la producción de dichos componentes.

4.10. Bus CAN

canAnalyser3 mini

Herramienta privada desarrollada para la utilización de un *hardware* concreto (USB-to-CAN), la cual ha sido utilizada para la generación de *Frames CAN* para la realización de las pruebas y los test correspondientes durante el desarrollo de la aplicación.

Se puede encontrar más información en el siguiente enlace:

<https://www.ixxat.com/products/products-industrial/tools/canalyzer/canalyzer-overview>

SocketCAN[14]

Se trata de un conjunto de herramientas y *drivers Open Source* para el bus CAN, desarrollado para *Linux*. Es posible la creación de interfaces CAN virtuales o añadir *hardware CAN* como si de una tarjeta de red se tratase.

Can-Utils

Se trata de un conjunto de herramientas con las cuales podemos interactuar con el bus CAN. A continuación se describen las dos herramientas utilizadas y más interesantes.

Puede encontrarse en el siguiente enlace: <https://github.com/linux-can/can-utils>

candump

Utilidad perteneciente al paquete de *can-utils*. Nos permite capturar todo el tráfico que es transmitido a través del bus y almacenarlo en un fichero, entre otras opciones.

canplayer

Utilidad perteneciente al paquete *can-utils*. Esta herramienta nos permite generar *Frames* y enviarlas al bus CAN. También es posible enviar un fichero capturado anteriormente con la herramienta *candump*.

Aspectos relevantes del desarrollo del proyecto

A continuación se tratan los aspectos más importantes que influyeron en el desarrollo del proyecto.

5.1. Inicio del proyecto

La idea de desarrollar esta aplicación surgió unos meses atrás, tras trabajar en una empresa que se dedicaba a la reparación de maquinaria.

Tras investigar y leer algunos manuales de reparación de esas máquinas, era de destacar que existía una característica en común, que era el uso del bus CAN en muchas de ellas, por no decir todas. A partir de ahí surgió el interés y las ganas de indagar sobre su funcionamiento.

Una vez fue posible la lectura de dichos datos, quedaba clara la dificultad para la identificación de los mismos, ya que todos ellos estaban en hexadecimal, lo cual dificultaba su identificación.

Tras el desarrollo de unos pequeños *scripts* en *Python*, los cuales eran capaces de realizar una gráfica con los datos previamente adquiridos del bus CAN y detectar la mayor facilidad a la hora de identificar esos datos, surgió la idea de realizar este proyecto.

Existía la necesidad de que, al contrario que en el caso anterior, este proyecto permitiera realizar ese proceso en tiempo real, no con una muestra de datos tomada anteriormente.

Otra de las principales motivaciones fue el hecho de que si, por ejemplo, existiera la necesidad de recoger los datos de un sensor (de velocidad de una

rueda) para su monitorización y este dato es transmitido a través del bus CAN, no es necesaria la instalación de un nuevo sensor, abaratando costes y facilitando realizar la tarea necesaria.

5.2. Metodologías

Se decidió utilizar la metodología *Scrum*, una metodología ágil, de forma que no era necesario definir el proyecto desde el principio (no se conocía el alcance del mismo, ni si iba a ser posible realizar ciertas propuestas planteadas).

La duración aproximada de los *sprints* era de una semana, a la vez que al finalizar dicho *sprint*, se planificaba el de la semana siguiente.

A través del uso de la herramienta *ZenHub*, se generaban *issues*, es decir, tareas a realizar en ese *sprint*, y se cambiaban de estado según se iban finalizando.

5.3. Formación

El proyecto requería conceptos tanto de funcionamiento como de la implementación del bus CAN, los cuales desconocía en un principio. A través de los siguientes documentos, se adquirieron conocimientos sobre la materia:

- *CAN Specification 2.0* [1].
- *Introduction to the Controller Area Network (CAN)* [7]
- *Controller Area Network (CAN) Implementation Guide*[3]
- *Bosch Controller Area Network (CAN) Protocol Standard*[13]
- *Data Communication in the Automobile*[17]
- *CAN Protocol Tutorial*[8]

5.4. Desarrollo del *hardware*

Existieron principalmente dos motivaciones para el diseño y desarrollo de un *hardware* propio:

- El elevado precio de las interfaces que existen actualmente en el mercado, las cuales pueden llegar a costar miles de euros.
- La posibilidad de conectarse a dos buses distintos de forma simultanea. Una interfaz CAN únicamente puede conectarse a un bus de datos. Existen alternativas más baratas como *USBtin*[4], pero que únicamente permiten conectarse a un bus.

Con este desarrollo propio, se consigue la conexión a dos buses de datos de forma simultánea y económica.

Se decidió comenzar por esta fase por los siguientes motivos:

- No era seguro que el *hardware* fuera a funcionar con lo que no se podía crear una dependencia por parte del software en caso de que esa parte no se realizase satisfactoriamente.
- La producción de la PCB requiere de unas semanas desde que envías el pedido hasta que el material llega.
- Era la parte más desconocida y en la cual habría que invertir más tiempo para aplicar la metodología sugerida a la hora del desarrollo de este tipo de proyectos.

Se podrían definir 4 etapas las cuales se han llevado a cabo para el desarrollo de este *hardware*:

Diseño del esquema eléctrico

El primer paso para el desarrollo de una placa de circuito impreso, es diseñar el esquema en el que se incluyen los componentes, y la conexión que se va a realizar entre ellos. En este caso, el objetivo era duplicar un esquema existente basado en *USBtin*[4], introduciéndole un *Hub USB*[9], todo ello en la misma placa.

En este esquema se especifican los componentes utilizados (resistencias, condensadores, circuitos integrados, etc). Además se define que entradas y salidas de esos componentes van conectados al resto de componentes.

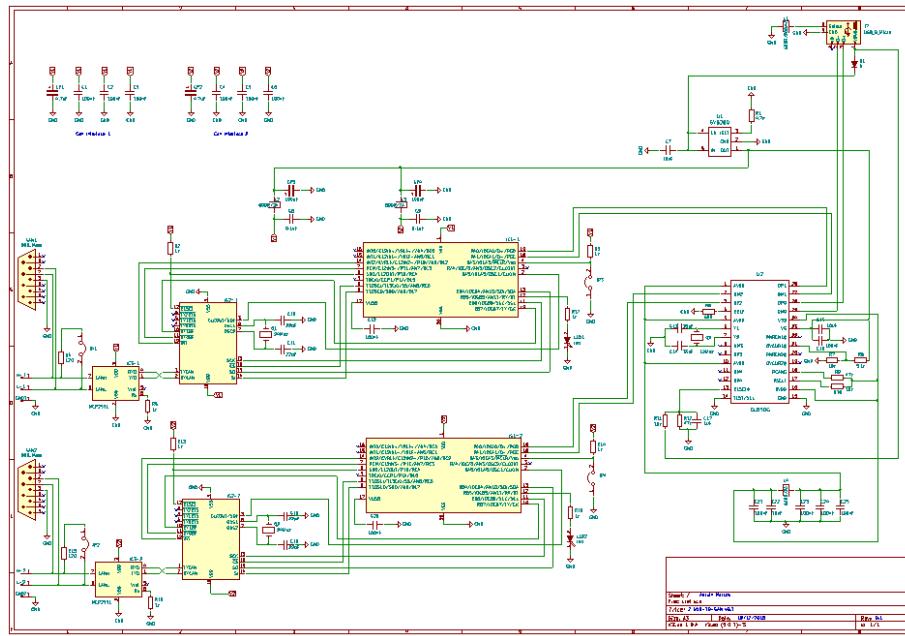


Figura 5.16: Vista general del esquema.

Generación del *netlist*

Una vez realizado el esquema eléctrico, necesitamos generar el fichero *Netlist*. En este fichero se almacenan los siguientes datos:

Por cada componente:

- Se describen las conexiones de cada uno de los elementos del circuito.
- El tipo de elemento que es cada componente.
- La huella (*footprint*) que va a utilizar, es decir, en función de cada encapsulado (forma física del *chip*), será necesario que la zona de soldadura de la PCB sea adaptada al mismo, tanto en tamaño como en forma.

Posicionamiento y *enrutado* de los componentes y las pistas en la PCB

Una vez se ha generado el fichero *Netlist*, este es cargado en el editor de PCBs.

En este momento, es cuando tenemos que decidir:

- El número de capas que va a tener nuestra PCB: En este caso se utilizaron dos capas al tratarse de un proyecto sencillo, pero en el que necesitaban hacerse cruces de pistas.
- La posición de los componentes y el *enrutado* de las pistas.

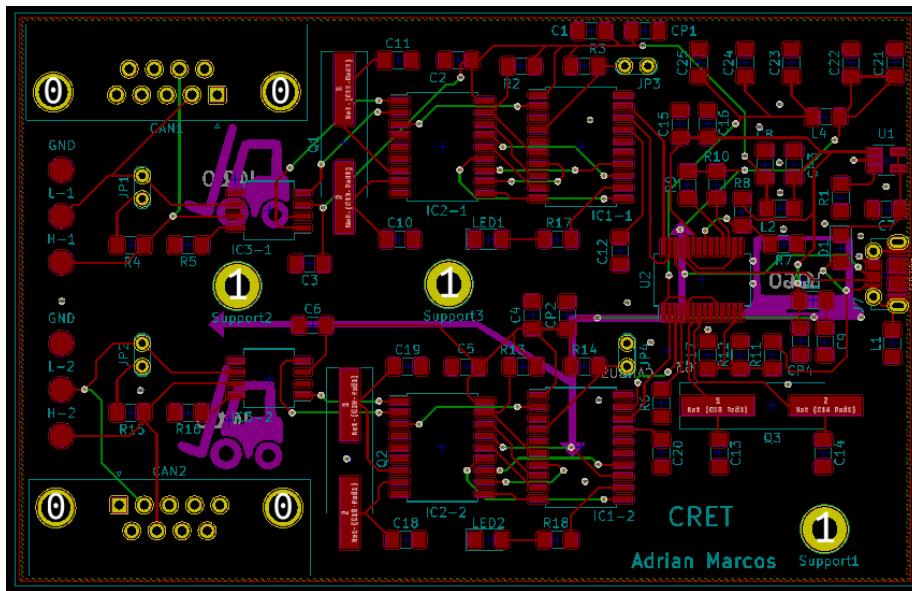


Figura 5.17: Esquema de la PCB del proyecto.

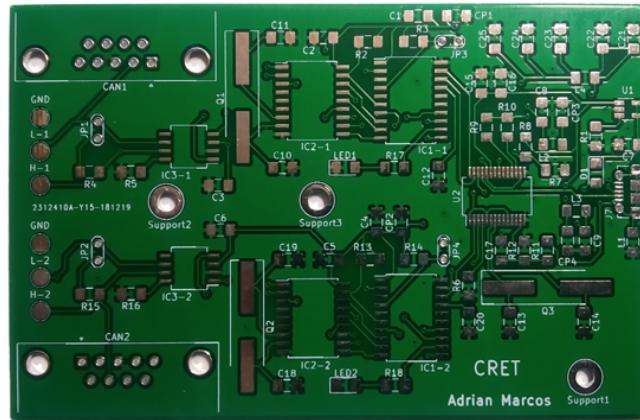


Figura 5.18: PCB del proyecto sin los componentes.

Tras limpiar la PCB, posicionar y soldar los componentes, obtenemos el siguiente resultado:

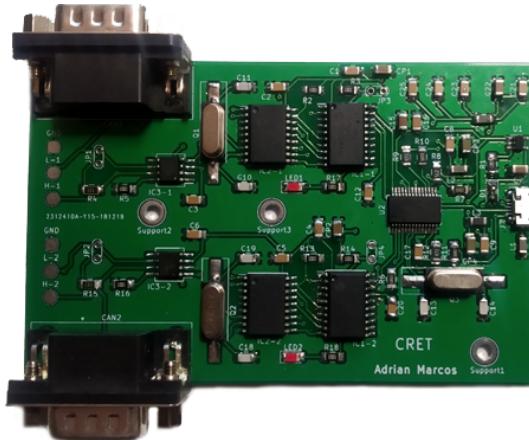


Figura 5.19: PCB del proyecto con los componentes soldados.

Carga del *firmware* en los IC

Uno de los problemas que surgieron una vez terminado el *hardware* fue la carga del *firmware*^[5] en los microcontroladores PIC.

Para realizar la carga del *firmware*, fue necesario desoldar ambos componentes y conectarlos a un programador para este tipo de dispositivos (*PICkit3*^[10]).

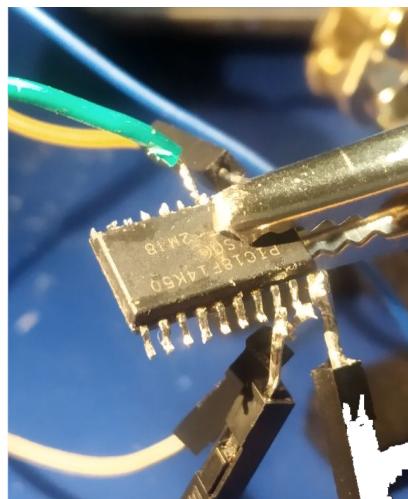


Figura 5.20: Pines conectados al microcontrolador PIC para la carga del *firmware*.

Para cargar el *firmware* fue necesario el programa *MPLAB X IPE v5.10*[12]

Conectando los pines del *PICkit3* al *PIC18F14K50* siguiendo su *datasheet*[11] fue posible la carga del *firmware* con éxito.

5.5. Desarrollo del *software*

Lo primero que había que decidir era el lenguaje en el que se iba a desarrollar la aplicación. Se barajó la posibilidad de realizarlo en Java o Python. La decisión de Java se debió a la disposición de una librería compatible con el *hardware* y a la librería JavaFX la cual nos permite realizar una aplicación muy visual y amigable de forma sencilla y eficiente.

Una vez elegido el lenguaje de desarrollo, se comenzó con la modelación de la interfaz gráfica, es decir, de la estructura de la aplicación a nivel visual. Una vez estaba claro, se comenzó con el desarrollo del código.

Para la base de datos, se decidió utilizar SQLite, ya que no requiere de ningún motor de base de datos, sino simplemente la utilización de una librería interna en la aplicación. Además, la información almacenada iba a ser muy pequeña y era más que suficiente.

Principalmente surgieron dos grandes problemas durante el desarrollo de la parte del software, ambos relacionados con la parte de la interfaz gráfica.

Generación de gráficas de forma dinámica

Era necesario que las gráficas se fueran creando de manera dinámica cuando existiera un nuevo dato a representar, no tener una estructura estática creada previamente. Para ello, cada vez que un nuevo *Frame* del bus CAN fuera detectado, había que proceder a la creación de dichos elementos, así como de un *listener* en ese objeto, el cual consultase la estructura en la que son almacenados los datos, para su posterior representación gráfica.

Para ello se decidió utilizar una estructura gráfica de tipo *GridPane*, dividiendo la pantalla en el eje horizontal en 3 celdas. Cada una de estas celdas se iría llenando en función de las necesidades e iría incrementándose de arriba a abajo si fuera necesario. Para conseguir ese efecto, el elemento padre del *GridPane* tenía que ser un *ScrollPane* es decir, lo que conocemos como un *Scroll*.

Todo esto tenía que ir de la mano con una serie de estructuras, también generadas dinámicamente, para realizar la lectura y escritura de los datos que tienen que ser representados.

De esa manera, se decidió utilizar estructuras de tipo *HashMap*, las cuales identificaban el dato por el par *ID* y número de *byte* representado.

Problemas de concurrencia

Los elementos que representan gráficamente los datos (*LineChart*), realizan una lectura de un tipo de datos llamado *ObservableList*. Durante la ejecución de la aplicación existen dos *threads*, uno para la ejecución de la parte gráfica, y el otro para la lectura de los datos del bus CAN.

El problema surge cuando se intenta realizar una escritura en el objeto de tipo *ObservableList* desde el *thread* que lee el bus CAN al mismo tiempo que el *thread* de la interfaz gráfica realiza una lectura de los datos del mismo objeto.

Para la resolución de este problema se decidió utilizar un modelo de tipo Productor/Consumidor. Para ello se definió un objeto de tipo *ConcurrentLinkedQueue* en el cual el *thread* encargado de la lectura del bus CAN, iba introduciendo la información que le llegaba al tiempo que el *thread* de la interfaz gráfica iba cargando esos datos en el objeto de tipo *ObservableList*. De esta manera, se consiguieron solucionar los problemas de concurrencia.

5.6. Diseño de la interfaz gráfica

Para el desarrollo de la interfaz gráfica, se ha utilizado la herramienta *SceneBuilder*.

Esta herramienta nos permite crear interfaces gráficas de forma fácil e intuitiva. Una vez realizado el diseño, la herramienta genera un fichero FXML (basado en una estructura XML), el cual será utilizado para mostrar los elementos en pantalla.

En este fichero es importante tener en cuenta los siguientes aspectos:

- Debemos indicarle qué controlador se va a encargar de manejar esa vista. Es decir, siempre tiene que existir una relación vista-controlador.
- Todos los componentes que vayan a ser utilizados desde el controlador, tiene que tener como mínimo la descripción *fx:id* para su identificación. Esta id tiene que ser única en el documento FXML.
- Además, en este fichero se indican las acciones que queremos que tengan los elementos. Por ejemplo, que al pulsar un botón o modificar un texto, el controlador ejecute una función establecida.

5.7. Testing

Para realizar las pruebas correspondientes durante el desarrollo de la aplicación y no depender del acceso físico a una máquina o vehículo, se ha desarrollado una pequeña placa la cual simularía una red CAN, en la que se pueden conectar dos nodos, uno para enviar datos al bus y el otro para recibir.

Para ello simplemente se han colocado dos resistencias de 120 *ohms* (definidas por el estándar del CAN), y unos conectores de tipo DB9, ya que son los más habituales en este tipo de *hardware*.



Figura 5.21: Placa casera para simular una red CAN.

5.8. Diseño e impresión 3D de una caja protectora

Para la protección de *hardware* desarrollado, se vio la necesidad de crear una caja la cual almacene dentro la placa, pero que además permitiera la conexión de los elementos necesarios para su funcionamiento.

Para ello se ha utilizado el *software Blender* para el diseño de la misma.



Figura 5.22: Renderizado de la caja en *Blender*.

Una vez concluido el diseño, el fichero 3D ha sido introducido en *Ultimaker CURA*, un *slicer* el cual nos transforma el objeto 3D en un fichero el cual la impresora 3D sea capaz de interpretar.

Para realizar la impresión, se ha utilizado una impresora 3D *BQ Prusa i3 Hephestos* y plástico de tipo ABS (*Acrylonitrile Butadiene Styrene*) el cual resulta más difícil de imprimir, pero es adecuado para un uso intensivo como puede ser el de la industria, teniendo más resistencia a golpes y a temperatura que uno de sus competidores como puede ser el PLA (*Polylactic acid*).



Figura 5.23: Caja impresa en 3D cerrada.



Figura 5.24: Caja impresa en 3D abierta.

Trabajos relacionados

Cada fabricante dispone de sus propias herramientas (al igual que muchos de ellos disponen de su propio *hardware*). Estas herramientas, como se ha mencionado en la introducción, tienen un coste muy elevado en la mayoría de los casos.

Las principales ventajas de este proyecto son:

- El *hardware* desarrollado puede ser producido por cualquier persona, ya que los esquemas y los ficheros necesarios para su producción son *open source*.
- Aplicación multiplataforma, compatible tanto con sistemas Windows como Linux.
- No es necesaria la instalación de *drivers* en el equipo en el que se va a utilizar la herramienta.
- Es la única herramienta libre que permite graficar los datos que fluyen por el bus CAN, así como su identificación y posterior monitorización.

Las principales desventajas son:

- Actualmente solo funciona con un *hardware* en concreto. Como se menciona en la sección Líneas futuras de trabajo 7.2, el desarrollo de este proyecto continuaría con la incorporación de un módulo para el soporte de los *drivers* *SocketCAN* en sistemas *Linux*.

A continuación se realiza una pequeña comparación entre las herramientas:

Herramientas	CRET	CANalyzat0r	PCAN Explorer
Visualización de los datos en RAW	X	X	X
Multiplataforma	X	X	
Hardware libre	X	N/A	
Representación gráfica de los datos	X		X
Etiquetado con un <i>click</i>	X		
Importar y exportar proyectos JSON	X		

Tabla 6.1: Comparación entre CRET y otras herramientas del mercado

Cabe destacar la presencia del proyecto *CANalyzat0r*[16], el cual nos permite también el análisis de los datos que pasan por el bus CAN, pero que no nos permite una rápida identificación a través de la visualización de los datos.

Además a través de CRET, se facilita mucho la identificación de las señales, pudiendo etiquetarlas y almacenarlas en el *dashboard*, y a su vez en una base de datos, para su utilización más adelante. En *CANalyzat0r*, el etiquetado de los datos identificados tiene que hacerse introduciendo la ID específica, además de que no nos permite separar los *bytes* del campo de datos.

Conclusiones y Líneas de trabajo futuras

7.1. Conclusiones

Una vez concluido el proyecto, es posible extraer las siguientes conclusiones:

- Se ha cumplido el objetivo general del proyecto. Ahora es mucho más fácil realizar un análisis de las señales que fluyen por el bus, así como su monitorización.
- Para conseguir el desarrollo del proyecto, ha sido necesaria la adquisición de nuevos conocimientos tanto en el ámbito del software, con el uso de JavaFX y la profundización en el lenguaje Java, así como a nivel de *hardware*, en el que se han adquirido conocimientos básicos pero suficientes para alcanzar el objetivo.
- Se han utilizado diversas estructuras generadas de forma dinámica y las cuales permiten acceso de forma concurrente a sus datos por varios *threads*. Esto ha ayudado a profundizar sobre el funcionamiento de este tipo de estructuras.
- Se han utilizado gran cantidad de tecnologías durante el desarrollo del proyecto, desde el diseño de elementos con *software* de modelado 3D pasando por el diseño y producción de *hardware* y el desarrollo de la aplicación en Java con el uso de librerías como JavaFX y Medusa para mejorar la interacción con el usuario.

7.2. Líneas de trabajo futuras

Es interesante proseguir con el desarrollo de la herramienta y adaptarla para su utilización con los módulos de *Linux SocketCAN*. De esta manera, se conseguiría deshacer de la dependencia que existe actualmente con el *hardware*, para la utilización de la aplicación. *SocketCAN* haría de intermedio entre el *hardware* y el *software* a modo de *driver* de todos aquellos dispositivos que sean soportados.

Otra de las ventajas de realizar esa modificación, sería la posibilidad de crear y utilizar interfaces virtuales dentro del equipo, de manera que ni siquiera sería necesario un *hardware* físico para realizar pruebas.

Bibliografía

- [1] BOSCH. Can specification, 1991. URL <http://esd.cs.ucr.edu/webres/can20.pdf>.
- [2] BOSCH. Osi layers in automotive networks, 2013. URL <http://www.ieee802.org/1/files/public/docs2013/new-tsn-diarra-osi-layers-in-automotive-networks-0313-v01.pdf>.
- [3] Analog Devices. Controller area network (can) implementation guide, 2017. URL <https://www.analog.com/media/en/technical-documentation/application-notes/AN-1123.pdf>.
- [4] Thomas Fischl. Usbtin - usb to can interface, 2014. URL <https://www.fischl.de/usbtin/>.
- [5] Thomas Fischl. Bootloader usbtin, 2017. URL <https://www.fischl.de/usbtin/#bootloader>.
- [6] Marco Guardigli. Hacking your car, 2012. URL <https://marco.guardigli.it/2010/10/hacking-your-car.html>.
- [7] Texas Instruments. Introduction to the controller area network (can), 2016. URL <http://www.ti.com/lit/an/sloa101b/sloa101b.pdf>.
- [8] Kvaser. Can protocol tutorial, 2016. URL <https://www.kvaser.com/can-protocol-tutorial/>.
- [9] Ltomov. Osh 4 port usb hub, 2016. URL <https://github.com/ltomov/4-port-usb-hub>.

- [10] Microchip. Pickit 3 in-circuit debugger, 2013. URL <https://www.microchip.com/Developmenttools/ProductDetails/PG164130>.
- [11] Microchip. Pic18(l)f1xk50, 2015. URL <http://ww1.microchip.com/downloads/en/devicedoc/40001350f.pdf>.
- [12] Microchip. Mplab ide, 2018. URL <https://www.microchip.com/mplab/mplab-x-ide>.
- [13] NXP. Bosch controller area network (can) protocol standard, 2010. URL https://www.nxp.com/files-static/microcontrollers/doc/data_sheet/BCANPSV2.pdf.
- [14] Volkswagen Research. Socketcan, 2010. URL <https://www.kernel.org/doc/Documentation/networking/can.txt>.
- [15] Aron Sreder. How to easily implement an application pre-loader, 2015. URL <https://blog.codecentric.de/en/2015/09/javafx-how-to-easily-implement-application-preloader-2/>.
- [16] sw pschmied. Canalyzat0r, 2018. URL <https://github.com/schutzwerk/CANalyzat0r>.
- [17] Vector. Data communication in the automobile, 2006. URL https://vector-academy.com/portal/medien/cmc/press/PTR/SerialBusSystems_Part2_ElektronikAutomotive_200612_PressArticle_EN.pdf.
- [18] Wikipedia. Slicer (3d printing), 2019. URL [https://en.wikipedia.org/wiki/Slicer_\(3D_printing\)](https://en.wikipedia.org/wiki/Slicer_(3D_printing)).
- [19] Wikipedia. Can bus - wikipedia, 2019. URL https://en.wikipedia.org/wiki/CAN_bus.