Facultad de Ingeniería
de la Universidad de la República

Proyecto final de grado de Ingeniería Eléctrica

Sistema de Alimentación para Vehículos Eléctricos

Desarrollo del unidireccional y estudio del bidireccional

Autores:
Mauricio Gutiérrez
Francisco Halty
Gustavo Mango

Tutores:
Ing. Federico Arismendi
Ing. Juan Pedro Carriquiry
Dr. Ing. Mario Vignolo

16 de octubre de 2020

AGRADECIMIENTOS.

Durante el transcurso del proyecto, fue imprescindible el asesoramiento y la colaboración del grupo ***Proyecto de Movilidad Eléctrica*** de UTE, por lo que se agradece especialmente al Ing. Pablo Cabalo, Ing. Marcelo Battagliese, Ing. Diego Bentancur e Ing. Fernando Ron. Se agradece a los tutores y a cada docente a lo largo de nuestra carrera, por la dedicación y el conocimiento generosamente brindado. A nuestros seres queridos, por el apoyo, la paciencia y los consejos en los momentos más importantes. Sin su ayuda, este proyecto no hubiese sido posible.

RESUMEN.

El presente documento "**Sistema de Alimentación para Vehículos Eléctricos**" se presenta como el Proyecto de Fin de Grado correspondiente a la carrera de Ingeniería Eléctrica - opción Potencia. Consta de dos objetivos independientes: el desarrollo del cargador convencional y un análisis teórico sobre cargadores bidireccionales.

Comienza con una descripción del Sistema de Alimentación del Vehículo Eléctrico (SAVE) que fue diseñado y se mencionan las características de los SAVE convencionales. Luego se presenta el protocolo OCPP 1.6 y se explica el código basado en el mismo e implementado en una placa electrónica Raspberry, para la comunicación entre el SAVE y el servidor del Centro de Control de Carga. También se presenta la norma IEC 61851-1 que define, en su anexo A, el protocolo de comunicación entre el SAVE y el vehículo eléctrico (VE). Luego se detalla un análisis del circuito implementado para cumplir dicho protocolo, y se explica el código basado en el protocolo antedicho e implementado en la placa electrónica Arduino. A continuación, se presentan los planos del circuito de potencia y del circuito de control, junto con la alimentación y la conexión de otros componentes en el SAVE. Se hace una síntesis de las pruebas realizadas y se muestra un instructivo para el uso del SAVE.

Finalmente, se realiza una revisión bibliográfica, del estado del arte y descripción teórica sobre los cargadores bidireccionales (aquellos que permiten la inyección de energía eléctrica desde el vehículo eléctrico hacia la red eléctrica).

Como conclusión: se evalúan funcionalidades del SAVE diseñado, se compara el SAVE diseñado con SAVEs comerciales, se evalúan tiempo-costo de fabricación y posibles mejoras al diseño implementado. Además se analiza la posibilidad de que el SAVE implementado pueda adaptarse para cumplir la funcionalidad bidireccional.

ÍNDICE GENERAL

GLOSARIO.

- VE: Vehículo Eléctrico.
- SAVE: Sistema de Alimentación de Vehículos Eléctricos (o cargador de VE).
- Centro de Control de Carga: El servidor que gestiona la carga de un SAVE o varios.
- OCPP 1.6: Protocolo de comunicación entre el SAVE y centro de control de carga.
- JSON: Objeto en Notación JavaScript. Es un formato de texto, para envío de datos.
- PDU: Mensaje en formato JSON con los campos determinados en OCPP 1.6
- IEC 61851-1: Protocolo de comunicación entre SAVE y Vehículo Eléctrico.
- SAE J1772: Protocolo de comunicación entre SAVE y Vehículo Eléctrico.
- G2V: Grid-to-Vehicle (red a vehículo).
- V2G: Vehicle-to-Grid (vehículo a red).
- V2H: Vehicle-to-Home (vehículo a hogar).
- V2L: Vehicle-to-Load (vehículo a carga eléctrica).
- OBC: On-Board Charger (convertidor a bordo del vehículo eléctrico)
- ISO/IEC 15118: Protocolo de comunicación entre SAVE y Vehículo Eléctrico.
- SECC: Controlador de comunicación del SAVE
- EVCC: Controlador de comunicación del VE
- PLC: Power Line Communication (comunicación mediante líneas de potencia)

CAPÍTULO 1

INTRODUCCIÓN.

1.1. Motivación.

En Uruguay, hasta mediados del 2018, "se comercializaban 13 modelos de 6 marcas diferentes de autos totalmente eléctricos y circulaban aproximadamente 160, de los cuales 90 eran las Renault Kangoo ZE que pertenecen a UTE, 24 BYD e6 y e5 de la flota de taxis, 45 de la marca eMin que comercializa Ruffino Group, una JAC S2 que utilizaba Brian Lempert director de Grupo Fiancar, 3 ejemplares Tesla importados por reconocidos empresarios uruguayos y un ómnibus eléctrico BYD K9 de Cutcsa." (ver [1]).

A principios de 2019, ya había 54 taxis eléctricos circulando y la Intendencia de Montevideo hizo un acuerdo, en abril de dicho año, para subsidiar la compra de 30 ómnibus eléctricos para el transporte urbano (ver [2] y [3]). Sabiendo que los vehículos eléctricos ya son una realidad en el Uruguay, es obvia la necesidad de cargadores de baterías para los mismos. Es por eso que la empresa proveedora de energía eléctrica en Uruguay UTE está instalando la denominada "Ruta Eléctrica", que es la primera red de cargadores de vehículos eléctricos en Latinoamérica.

Por otra parte se tiene como objetivo municipal, en Montevideo, comenzar el año 2020 con 300 taxímetros con movilidad eléctrica, un 10 % de la flota total. Y como objetivo nacional, alcanzar una flota de 100 ómnibus eléctricos en el 2021. (ver [2] y [3]). Con una flota de vehículos en aumento, es de interés estudiar las posibles variantes de cargadores para satisfacer la demanda de manera eficiente.

La motivación de implementar el cargador convencional, es poder conocer en detalle los componentes del mismo y el software encargado de la comunicación entre: Centro de Control de Carga-Cargador y Vehículo Eléctrico-Cargador. Como hasta ahora se importaron los cargadores ya fabricados, es posible que a nivel local se esté desaprovechando oportunidades de optimizar el sistema de cargadores. Mientras que el motivo del estudio de cargadores bidireccionales, es por conocer la posibilidad de una mejora tecnológica que permita utilizar la energía almacenada en las baterías del vehículo como generación distribuida.

1.2. Objetivos.

El objetivo principal es diseñar y construir un Sistema de Alimentación de Vehículo Eléctrico (SAVE) que permita cargar la batería de un vehículo eléctrico (VE), a partir de la potencia eléctrica suministrada por la red, la funcionalidad denominada Grid-to-Vehicle (G2V).

El segundo objetivo es describir los requerimientos necesarios para que el SAVE diseñado, además de funcionar en la forma convencional G2V, también soporte y ejecute de forma controlada la funcionalidad inversa, Vehicle-to-Grid (V2G), la que utiliza la energía almacenada en la batería del vehículo para transferirla hacia la red eléctrica.

1.3. Alcance.

El principal desafío, del primer objetivo, es que el SAVE se tendrá que comunicar con el vehículo eléctrico (VE) para suministrar la energía eléctrica en el momento adecuado. Y a su vez, el SAVE se tendrá que comunicar con el Centro de Control de Carga, que controlará toda la sesión de carga del VE. Se realizarán las descripciones de cada comunicación, el código que ejecuta dichas comunicaciones, los planos constructivos del SAVE, memorias de cálculo, memorias técnicas y simulaciones mediante software. Finalmente, se construirá el cargador y se ensayará su funcionamiento utilizando un vehículo eléctrico. La instalación del cargador, en un punto de carga fijo, está fuera del alcance del proyecto.

Para el segundo objetivo, se estudiarán los distintos diseños posibles de un cargador bidireccional y se analizarán ventajas y desventajas de cada diseño. Se analizan las posibles modificaciones al SAVE convencional, desde las comunicaciones necesarias para controlar la funcionalidad V2G, como los conectores que soportan dicha comunicación. Queda fuera del alcance: el diseño del circuito de control, el desarrollo del software de comunicación y la implementación del cargador bidireccional.

CAPÍTULO 2

DESCRIPCIÓN DEL SISTEMA.

El Sistema de Alimentación de Vehículo Eléctrico (**SAVE**) diseñado, tiene como objetivo, suministrar energía eléctrica desde la red hacia el vehículo, de forma controlada. Para lograrlo, el SAVE cuenta con un circuito de control, que habilita dicho suministro solo cuando el vehículo eléctrico (VE) se encuentre en condiciones aptas para recibir dicha energía. Aunque el VE esté conectado al SAVE, no alcanza para determinar si está listo para ser cargado. Para que el SAVE reconozca en qué momento habilitar el suministro, existen protocolos que definen una comunicación entre el SAVE y el VE. También para cada protocolo, existe un tipo de conector diseñado para habilitar dicha comunicación.

Por otro lado, es usual gestionar varios SAVEs, mediante una comunicación entre cada SAVE y un servidor que funcione como un **Centro de Control de Carga**. Este servidor, podrá establecer el horario en que cada SAVE está habilitado para una sesión de carga, verificará si el usuario del vehículo tiene permiso para cargar, almacenará datos medidos por el SAVE durante la sesión de carga y podrá reportar errores cuando los haya, entre otras tantas funciones. Por ejemplo, en el caso de que el servidor no habilite la sesión de carga de un SAVE, aunque el vehículo esté en condiciones de recibir la energía eléctrica, el SAVE no habilitará el suministro.

En la figura 2.1 se muestra un esquema que representa la comunicación del SAVE con el resto del sistema, durante una sesión de carga de vehículo eléctrico.

Figura 2.1: Esquema representando la comunicación entre el SAVE y el resto del sistema.

Para abordar la temática de la carga de VE, es necesario distinguir cuatro conceptos: los tipos de carga, los tipos de conexión, los modos de carga y los tipos de conectores.

2.1. Tipos de Carga

Los tipos de carga, hacen referencia directamente a cuánto tiempo se necesita para cargar la batería del vehículo; lo que está estrechamente relacionado al nivel de potencia suministrado.

- **Carga lenta**: Este tipo de carga se realiza en corriente alterna y la demanda de potencia en este caso es de $2,2\ kW$. Sirve para casos de emergencia, pero no para cargar completamente la batería del VE.
- **Carga estándar**: Este tipo de carga se realiza en corriente alterna y demanda una potencia entre $3,7\ kW$ y $7,4\ kW$. En este caso la carga de la batería de un vehículo podrá demorar toda la noche.
- **Carga semi rápida**: Este tipo de carga se realiza en corriente alterna y demanda una potencia entre $7,4\ kW$ y $22\ kW$. En este caso la batería del vehículo se cargará por completo entre seis horas y cuatro horas.
- **Carga rápida**: Esta carga se realiza a potencias entre $22\ kW$ y $43\ kW$, se realiza en corriente alterna. En este caso la batería del vehículo se cargará por completo entre cuatro y dos horas.
- **Carga súper rápida**: En este caso la carga se realiza en corriente continua y los valores de potencia son hasta $120\ kW$. En este caso la batería del vehículo se cargará por completo en menos de dos horas.

  El vehículo cargará a la potencia admisible, que estará limitada por la potencia máxima que suministra el SAVE o por la máxima potencia admisible por el cable (ver [4]).

Se decidió desarrollar un SAVE que suministre un **tipo de carga semi-rápida** de hasta $22\ kW$, por lo que el tiempo de carga es razonable para utilizar el SAVE en la vía pública. Además, al suministrar en **corriente alterna** (AC), no fue necesario diseñar un conversor AC/DC para integrarlo dentro del SAVE. Eso no genera un problema, porque es usual que los VE admitan suministros en corriente alterna, ya que tienen el conversor a bordo.

2.2. Tipos de Conexión.

- **Tipo de conexión A:** El VE tiene permanentemente unido un cable de alimentación que en el otro extremo tiene un conector para conectarse a la red eléctrica (o al SAVE).
- **Tipo de conexión B:** El VE se conecta utilizando un cable de alimentación desmontable, con un conector para el vehículo en un extremo y otro conector, en el otro extremo, para la conexión a la red eléctrica (o al SAVE).
- **Tipo de conexión C:** El VE se conecta utilizando un conector que tiene un cable de alimentación permanentemente unido a la red eléctrica (o al SAVE).

2.3. Modos de Carga.

Es necesario contar, en todos los modos de carga, con un dispositivo de control de corriente residual (llave diferencial) en conjunto con un dispositivo de protección contra sobrecorrientes.

- **MODO 1**: El VE se conecta directamente a la red eléctrica, utilizando un tomacorriente normalizado de hasta 16 A (monofásico o trifásico + conductor de tierra).
- **MODO 2**: El VE se conecta a la red eléctrica utilizando un tomacorriente normalizado de hasta 32 A (monofásico o trifásico + conductor de tierra), mediante con un conductor que posea un dispositivo que ejecute la comunicación con el VE a través del borne **Control Pilot** para indicar la máxima corriente configurada de forma permanente en el dispositivo.
- **MODO 3**: El VE se conecta a la red eléctrica mediante un SAVE que le suministre corriente alterna. El SAVE permanece conectado a la red eléctrica, y ejecutará la comunicación con el VE a través del borne **Control Pilot**, para indicar a corriente máxima regulada según el tipo de conector que se utilice para la carga del vehículo.
- **MODO 4**: Es igual al modo 3 con la salvedad que la carga es en corriente continua.

  Con el objetivo de suministrar corriente continua, es necesario que el SAVE tenga integrado un conversor AC/DC en su interior.

La siguiente imagen ilustra los cuatro modos de carga.

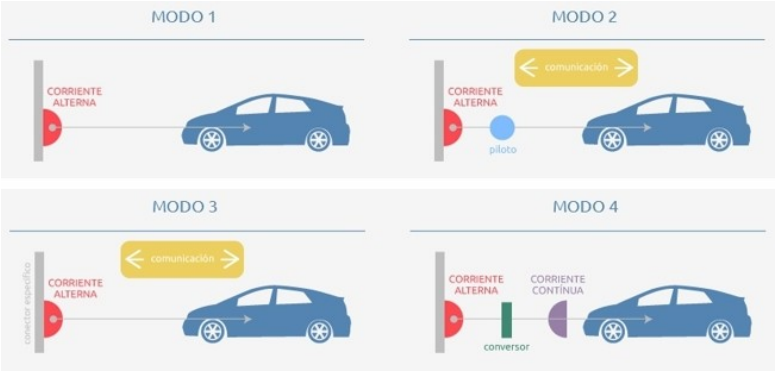


Figura 2.2: Modos de carga.

El SAVE diseñado ejecuta el **Modo 3** debido a que se eligió cargar en corriente alterna, sin utilizar un tomacorriente normalizado para la conexión a la red y suministrando una corriente máxima de 32 A pero regulada según el tipo de conector que utilice el usuario del vehículo. Una de las ventajas de implementar un SAVE, es que se puede realizar una comunicación con un servidor de **Centro de Control de Carga**.

En el caso del SAVE diseñado, se implementó un circuito de potencia que soporta hasta 32 A con un **tipo de conexión A o B**. Llegado el caso que se quiera realizar una carga de mayor potencia, alcanzaría con pocas modificaciones en el circuito de potencia y en el software del SAVE.

2.4. Tipos de Conectores.

Al existir distintos tipos de conectores, porque aún no se estandarizó uno para todos los fabricantes, en esta sección se presentan los más utilizados en diferentes países y se detallan sus principales características.

- El conector más básico que se encuentra en el mercado es el conector tipo schuko, el cual se rige con el estándar CEE 7/4 Tipo F. Este conector es compatible con los tomacorriente europeos. Por construcción presenta un toma de tierra y dos bornes, llegando a soportar hasta una corriente de 16 A. Además de estar en la gran mayoría de los electrodomésticos, es muy común encontrarlo en motocicletas y bicicletas eléctricas. Se usa para algunos coches eléctricos, como el Twizy. Este tipo de conector de un vehículo eléctrico se utiliza para una carga lenta del mismo y sin comunicación integrada.

- El conector Yazaki o Tipo 1 (protocolo SAE J1772), es el conector japonés estándar para la carga de vehículos eléctricos en corriente alterna, el cual también se adoptó en Estados Unidos. Como características físicas cuenta con un diámetro de 43 mm y en su interior se ubican cinco bornes: dos de potencia, uno para la tierra y dos bornes complementarios. Un borne complementario es el de detección de proximidad (**Proximity Pilot (PP)**), el cual chequea que el vehículo no se mueva mientras esté conectado y el otro es el borne de control (**Control Pilot (CP)**) se utiliza para la comunicación entre el SAVE y el vehículo eléctrico. Este conector Tipo 1, es utilizado para dos tipos de carga (o niveles, como se define en el protocolo SAE). El nivel 1, el cual admite una corriente de 16 A que se utiliza para un tipo de carga lenta de un vehículo eléctrico; y el nivel 2, que admite una corriente de hasta 63 A $(43,8\ kW)$ que se utiliza para el tipo de carga rápida. Este tipo de conector se puede encontrar en modelos como Opel Ampera, Nissan ENV200, Nissan Leaf, Mitsubishi iMiev, Mitsubishi Outlander o Peugeot iON.

Figura 2.3: Conector Tipo 1.

- El conector Tipo 2 (protocolo IEC 62196-2) o Mennekes (protocolo VDE-AR-E 2623-2-2) es un conector alemán de tipo industrial. Es uno de los más utilizados y está aprobado como estándar en Europa. Es un conector para corriente alterna que permite alimentar cargas trifásicas de hasta 63 A con una potencia de 44 kW o cargas monofásicas de hasta 16 A entregando una potencia de $3,5\ kW$.
  En comparación al conector Tipo 1: Ambos conectores tienen los mismos bornes **CP**, **PP** para la comunicación con el VE y fase, neutro, tierra para el suministro de potencia. Pero además el Tipo 2 incorpora dos clavijas extra que corresponden a las dos fases adicionales

necesarias para la carga trifásica. Este conector utiliza el protocolo IEC 61851-1, que define una comunicación entre el SAVE y el vehículo, compatible con la comunicación del Tipo 1 (SAE J1772). Este tipo de conector se puede encontrar en vehículos eléctricos como BMW i3, i8, BYD E6, Tesla Model S, Renault Zoe, híbrido enchufable Volvo V60, VW E-up, Audi A3 E-tron, plug-in Mercedes S500, híbrido enchufable VW Golf, Porsche Panamera y Renault Kangoo ZE.

Figura 2.4: Conectores Tipo 2.

- El conector GB/T (protocolo GB/T 20234.2) es el estándar de los fabricantes chinos. Es un conector que está diseñado para la carga rápida en corriente alterna (también hay otro para corriente continua). Es compatible con el Tipo 2 europeo porque tiene tres bornes de potencia, neutro, toma de tierra y dos bornes para la comunicación con el VE. .

Figura 2.5: Conector GB/T Type 2.

- El Conector conocido como CHAdeMO, es el estándar de los fabricantes japoneses (Mitsubishi, Nissan, Toyota y Fuji, de quien depende Subaru). Es un conector que está diseñado para la carga súper rápida en corriente continua. Constructivamente presenta 10 bornes, toma de tierra y comunicación con el VE.

Figura 2.6: Conectores CHAdeMO.

- El conector CCS Combo 1 o Combo 2 (protocolo IEC-62196-3) es una propuesta de norteamericanos y alemanes para proporcionar una solución estándar. Este tipo de conector es utilizado por fabricantes como Audi, BMW, Porsche y Volkswagen.

  El Combo 1 macho tiene dos bornes para conducir corriente continua, protección a tierra, Control Pilot y Proximity Pilot. Cuando la conexión es entre el Combo 1 macho y hembra, entonces se realiza la carga en corriente continua. Pero como el Combo 1 hembra, permite compatibilidad con el conector Tipo 1 macho, en caso de conectarse, se realiza una carga en corriente alterna.

  El Combo 2 macho tiene dos bornes para conducir corriente continua, protección a tierra, Control Pilot y Proximity Pilot. Cuando la conexión es entre el Combo 2 macho y hembra, entonces se realiza la carga en corriente continua. Pero como el Combo 2 hembra, permite compatibilidad con el conector Tipo 2 macho, en caso de conectarse, se realiza una carga en corriente alterna.

Figura 2.7: Conectores CCS combo 1

Figura 2.8: Conectores CCS combo 2

La siguiente imagen muestra cada tipo de conectores, según región y tipo de corriente:

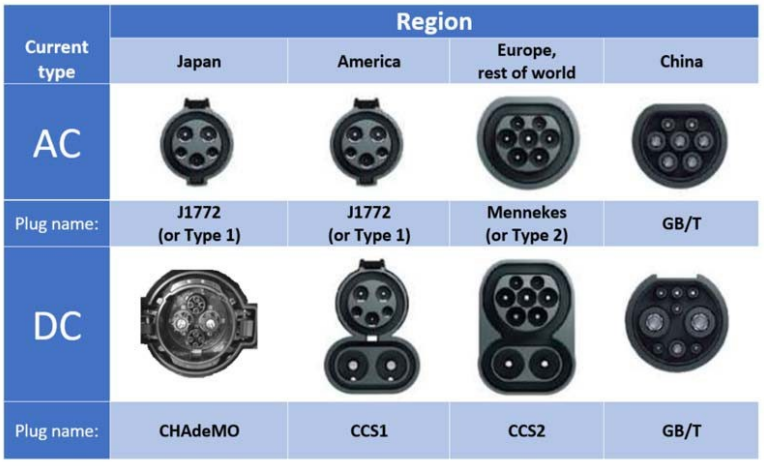


Figura 2.9: Tipos de conectores, según región y tipo de corriente.

La UNIT aprobó las normas europeas IEC 61851-1 y IEC 62196-2. La primera norma, en el Anexo A, define el protocolo de comunicación entre el VE y el SAVE. Mientras que la segunda define el conector Tipo 2, por lo cual es usual que en el mercado local se comercialicen vehículos eléctricos con dichos conectores.

Es por dicha razón que el SAVE diseñado implementa el **protocolo IEC 61851-1** para la comunicación entre SAVE y el vehículo eléctrico. Y tiene un **conector Tipo 2** hembra, para implementar el tipo de conexión A o B.

2.5. Resumen del Sistema.

El SAVE desarrollado está diseñado para poder cargar, en Modo 3, cualquier VE con un tipo de conexión A o B, que tenga un conector Tipo 2. El SAVE está compuesto por un medidor de energía, un circuito de potencia trifásico de 400 V, 22 kW, con llave termo-magnética, llave diferencial, un contactor para habilitar el suministro de potencia al VE y un relé que lo comanda. Los planos del circuito de potencia, del circuito de control, de la alimentación del medidor y de los componentes de electrónica; se presentan en el **capitulo 8**.

Con la electrónica se implementan dos comunicaciones en simultáneo:

Por un lado, utilizando una placa **Arduino** (descrita en el Anexo A.1), se ejecuta la comunicación entre el SAVE y el VE, en base a lo definido en el anexo A del **protocolo IEC 61851-1**. Mientras que, utilizando una placa **Raspberry**[1], se implementa la comunicación entre el SAVE y el Centro de Control de Carga, en base al **protocolo OCPP**. Se eligió este protocolo, realizado por la Open Charge Allience (OCA), porque es el más utilizado en el mercado y es de acceso libre (open source). Y se implementó la versión **OCPP 1.6**, fundamentalmente porque la empresa uruguaya UTE posee un servidor (denominado "CargaMe") que funciona como un Centro de Control de Carga y el grupo *Proyecto de Movilidad Eléctrica* de UTE permitió que el SAVE diseñado utilizara dicho servidor. Ambas comunicaciones serán descritas con mayor detalle en las siguientes dos secciones del presente capítulo.

En la figura 2.10 se muestran esquemáticamente los componentes del sistema y las funciones que cumplen: con la Raspberry se ejecuta la comunicación con el Centro de Control de Carga (servidor de UTE, en este caso), mientras que con el Arduino se ejecuta la comunicación con el vehículo y también se controla el relé que habilita el suministro de potencia al VE (mediante el comando del contactor). Dentro del SAVE, no se implementó un conversor AC/DC porque los vehículos que admiten una carga en corriente alterna ya lo tienen integrado a bordo.

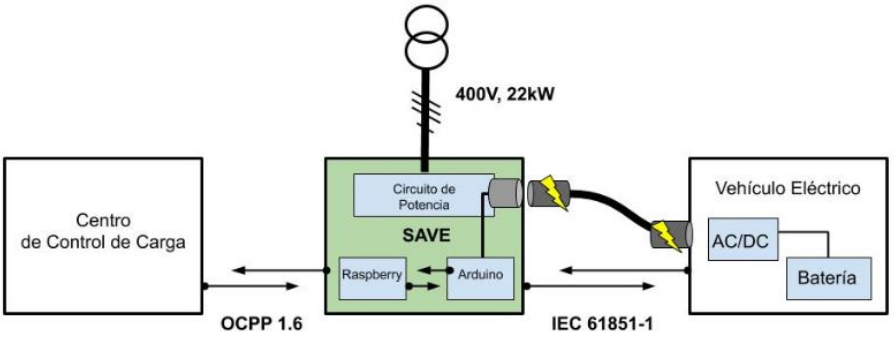


Figura 2.10: Diagrama de bloques del sistema.

Las pruebas finales del SAVE diseñado - ensayando la carga de un VE - y el instructivo de uso de dicho SAVE, se presentan en el **capítulo 9**. Mientras que, en el **capítulo 11**, se detallan las conclusiones sobre: la funcionalidad del SAVE implementado, el tiempo de fabricación y sus costos, una comparación entre el SAVE diseñado y los SAVE comerciales, junto con posibles mejoras del diseño.

[1]Raspberry: es una computadora de placa única (Single Board Computer), que puede realizar las mismas funciones básicas que una computadora. Tiene un sistema operativo, se le pueden instalar programas de software, etc; pero tiene limitaciones por restricciones de costo y espacio físico. Ver `https://www.raspberrypi.org/`

2.6. Comunicación entre el SAVE y el Servidor.

Esta comunicación inalámbrica se ejecuta entre una **Raspberry** dentro del SAVE y un servidor que funciona como un Centro de Control de Carga. La conexión es mediante una red WiFi VPN -una red privada con acceso restringido- y la Raspberry se conecta a dicha red, mediante una señal WiFi generada por un modem que está dentro del SAVE (ver sección **4.2.3**).

Se comenzó investigando posibles lenguajes de programación para implementar la comunicación entre el SAVE y el servidor del centro de control de carga. La característica principal era que el protocolo OCPP 1.6 establece el requerimiento de abrir un canal WebSocket (un canal donde los datos pueden entrar o salir sin una clara correlación) para permitir una comunicación en la que tanto el cliente (SAVE en este caso) como el servidor, puedan iniciar una solicitud (**request**) o responder a la solicitud correspondiente (**response**).

Como el servidor a utilizar, envía y espera recibir mensajes con un formato de texto sencillo denominado JSON (JavaScript Object Notation). Entonces, es necesario que el SAVE envíe mensajes con el mismo formato y una de las posibilidades era aplicar el lenguaje **NodeJS**, que está basado en **JavaScript**. En primera instancia, la elección de NodeJS se debió a que la comunicación WebSocket con el servidor es de fácil implementación y de forma compacta, a través de la librería WS. Otra de las razones, es que NodeJS es un software libre, por lo que, es de fácil acceso y se dispone de mucha información para su utilización.

Habiendo elegido NodeJS como el lenguaje a utilizar, se comenzó implementando el código en una laptop, hasta lograr la conexión Websocket con el servidor. Una vez lograda dicha conexión, se implementaron mensajes de solicitud (según el formato OCPP 1.6) y se obtuvieron respuestas del servidor a cada mensaje enviado desde la laptop. El formato y el contenido de cada mensaje JSON implementado, son detallados en el **capítulo 3**.

El siguiente paso fue desarrollar un código NodeJS, de acuerdo a la lógica de comunicación definida por el protocolo OCPP 1.6 (también descrito en el **capítulo 3**). En la lógica implementada, primero se envía un mensaje de solicitud al servidor, y en función del contenido de la respuesta, el código debe evaluar cuál es el siguiente mensaje de solicitud a enviarle al servidor. El código que ejecuta dicho algoritmo, se detalla en el **capítulo 4** y se ejecuta dentro de la Raspberry.

Como la conexión websocket fue inicialmente resuelta en una laptop, para reducir costos y espacio dentro del SAVE, fue necesario utilizar la Raspberry para sustituirla. La ventaja de elegir una placa como la Raspberry, es que funciona de forma análoga a una computadora y puede sustituirla, sin tener que modificar el código que se estaba ejecutando. Además, es importante notar que mediante el lenguaje JavaScript (que es compatible con NodeJS) es posible implementar de forma sencilla la comunicación con los componentes periféricos. Por lo que la comunicación entre la Raspberry y el medidor o la comunicación entre la Raspberry y el Arduino, no fueron un impedimento.

Por último, se observó que el protocolo OCPP 1.6 establece la posibilidad de que el SAVE funcione temporalmente en modo "offline"(sin comunicación con el servidor). Con la condición de que, cuando se restablezca la comunicación, el SAVE tendrá que enviar todos los mensajes JSON que debieron haber sido enviados al servidor. Aunque esta función no fue implementada en el SAVE diseñado, es importante notar que NodeJS es un lenguaje pensado para el manejo de base de datos (donde se podrían almacenar todos los mensajes JSON utilizados por cierto tiempo), y esto podría ser beneficioso a futuro, si se desea implementar el modo "offline" del protocolo OCPP 1.6, osea sin comunicación con el centro de control de carga.

2.7. Comunicación entre el SAVE y el Vehículo Eléctrico

El protocolo IEC 61851-1, en su Anexo A, define una comunicación entre el SAVE y el VE, a través del cableado del conector Tipo 2. Dicho protocolo establece que el SAVE genere una señal eléctrica cuadrada, una modulación de ancho de pulso (PWM), con ciertas características (ver imagen 2.11) para indicarle al VE la máxima corriente que el SAVE le puede suministrar. Y también establece que el SAVE mida el voltaje en el borne **Control Pilot** del conector, para determinar el estado en que el VE se encuentra.

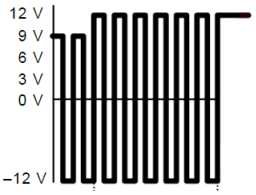


Figura 2.11: Forma de onda del PWM generado en el SAVE

Se decidió implementar dicha comunicación mediante un **Arduino UNO**, porque dicha plaqueta electrónica tiene pines de entrada analógicos, por lo que una medida de voltaje puede ser procesada sin dificultades. Además tiene pines de salida digital, que permiten generar una señal PWM con un ciclo de trabajo (**Duty Cycle**[2]) regulable. Para mayor información sobre Arduino Uno, ver el Anexo A.1.

Según el protocolo, el SAVE debe generar una señal PWM desde $-12\ V$ hasta $+12\ V$, con una frecuencia fija de 1 kHz y un ciclo de trabajo (**Duty Cycle**) regulable. Esta señal tiene que generarse en el circuito entre el borne **Control Pilot** (CP) y el borne de tierra de protección, de un conductor Tipo 2 ([3]) como se observa en la figura 2.12.

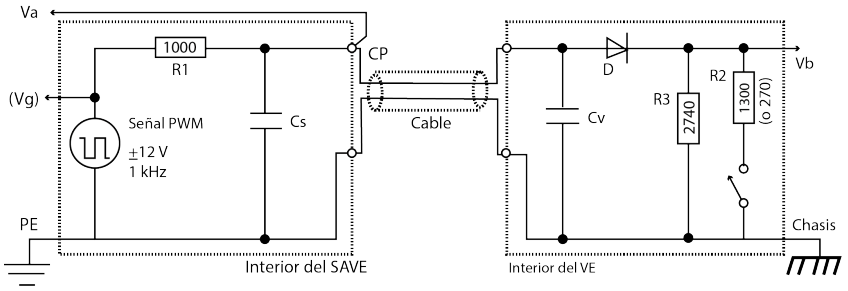


Figura 2.12: Representación del circuito donde se implementa la comunicación entre SAVE y VE (imagen del protocolo IEC 61851-1)

En la figura 2.12 se indica que el lado izquierdo del circuito estará en el interior del SAVE. Por lo que la señal PWM debe generarse dentro del mismo y la medida del voltaje V_a (realizada en este SAVE por el Arduino) servirá para detectar en qué estado se encuentra el VE.
Mientras que dentro del vehículo se pueden modificar valores de resistencia, swicheando la llave, para variar el voltaje (V_a) y así indicar que el VE cambió de estado.

[2]El Ciclo de trabajo (Duty Cycle) de una señal periódica y analógica se define como la relación entre el tiempo en que la señal está activa (en el valor máximo), con respecto al periodo total de la señal.

[3]Este mismo circuito también puede ser implementado en conectores Tipo 1 o CCS, porque los tres tipos de conectores tienen bornes Control Pilot compatibles con la misma comunicación mediante señal PWM.

En el **capítulo 5** se detallan los posibles estados del proceso de carga del VE (según el valor del voltaje V_a) y del SAVE (según si genera el PWM o no), como son definidos en el protocolo IEC 61851-1. También se menciona el criterio por el cual, el SAVE regula el Duty Cycle del PWM, para indicar cuál es la máxima corriente que suministrará.

Con el Arduino, dentro del SAVE, se genera una señal PWM desde 0 V hasta +5 V, con una frecuencia de 1 kHz y con el Duty Cycle regulado. Esta señal se amplifica, mediante un circuito diseñado para esta funcionalidad (analizado en el **capítulo 6**), para que la señal PWM llegue al VE con el rango de voltaje establecido en la norma.

Una vez generada esa señal PWM, el VE la detectará, y colocará cierto divisor resistivo para modificar el voltaje V_a y así indicar el estado del proceso de carga en el que se encuentra. El divisor resistivo afectará solamente el voltaje máximo de la señal PWM, debido a que a la entrada del VE hay un diodo. Dicho diodo funciona como rectificador de media onda, anula el lado negativo de la señal PWM, y por eso el VE no puede modificar dicha parte de la señal.

A partir de la medida del voltaje, el Arduino determinará el estado en que el VE se encuentra (ver **capítulo 5**). Cuando el VE esté disponible para la carga, el Arduino puede accionar un relé que energizará (o desenergizará) un contactor para cerrar (o abrir) el circuito de potencia que suministra la energía eléctrica desde la red hasta el VE. Es decir, que el Arduino comanda el circuito de control que habilita o deshabilita el suministro de energía eléctrica. El circuito de potencia y control se describe en el **capítulo 8**.

También es mediante el Arduino que se implementa la lectura de la tarjeta[4] de identificación de usuario para iniciar o detener una sesión de carga de VE. Pero una vez que se detecta una tarjeta, se envía el código de dicha tarjeta a la Raspberry (mediante un cable USB, ver sección 4.2.2) para que la Raspberry valide con el servidor (Centro de Control de Carga) si la tarjeta corresponde a un usuario habilitado para iniciar o detener la sesión de carga. La Raspberry recibirá la respuesta del servidor y se la enviará al Arduino; para que éste habilite (o no) la solicitud del usuario.

Por lo mencionado anteriormente, es notorio que el Arduino realiza ciertas acciones en base al voltaje que mide en el borne del **CP**, y en base a lo que recibe de la Raspberry. Este algoritmo es descripto en el **capítulo 7**. Fue implementado por el equipo, está guardado en la memoria del Arduino y el lenguaje es compatible con Arduino (que está basado en **C++**).

[4]Para la lectura de la tarjeta se utilizó un modulo de lector de tarjeta, RFID RC522, compatible con Arduino. Para más información ver `https://www.youtube.com/watch?v=LvRfxGTUEpE`

CAPÍTULO 3

PROTOCOLO OCPP: COMUNICACIÓN SAVE-SERVIDOR.

Para la comunicación entre un SAVE y el centro de control de carga, se utiliza el protocolo OCPP 1.6 (ver [5]) con mensajes en formato JSON (ver sección 3.2). El centro de control de carga actúa como servidor WebSocket (ver sección 3.2) y el SAVE actúa como un cliente WebSocket, por lo que es posible que la comunicación la inicie tanto el cliente como el servidor.

Usualmente, el cliente comienza la comunicación enviando un mensaje al servidor (**request**), y requiriendo una respuesta posterior (**response**). Pero una de las ventajas de utilizar WebSocket, a diferencia de otros protocolos (como el HTTP), es que el WebSocket permite que la comunicación la inicie tanto el servidor como el cliente.

Por lo tanto, sin hacer distinción de si es el cliente o el servidor:
El primer mensaje será un pedido o **request** (.req) con formato JSON, con ciertos campos obligatorios determinados por el protocolo OCPP (ver sección A.3). La contra-parte, al recibir dicho mensaje, primero valida los datos, se procesan los mismos y se envía una respuesta o **response** (.conf) con los campos obligatorios determinados por el protocolo OCPP 1.6. En base a los datos recibidos, el servidor o el SAVE ejecutan acciones: se envían otros mensajes JSON o se realizan medidas o se habilita la comunicación con el VE, etc.

Se presentarán los siguientes mensajes:

- Para notificar el reinicio del SAVE: **BootNotification**.
- Para solicitar una habilitación (para iniciar o detener una sesión): **Authorize**.
- Para enviar datos de medición: **MeterValues**.
- Para iniciar una sesión de carga: **StartTransaction**.
- Para detener una sesión de carga: **StopTransaction**.

Para poder explicar de mejor manera la comunicación entre el SAVE y el servidor se decidió desarrollar el presente capítulo de la siguiente forma. Primero se presentará brevemente la lógica entre los mensaje JSON tanto en el proceso de encendido del SAVE como de inicio (o parada) de una carga, pretendiendo generar en el lector una noción de como se lleva a cabo esa comunicación sin entrar en detalle de que contienen los mensajes, que define el valor de

cada campo dentro del mensaje, etc. Luego se describe en detalle la estructura de un mensaje JSON, se define WebSocket, etc. Luego se presenta un resumen de cada uno de los mensajes. El capítulo finaliza con cada uno de los mensajes que se intercambian en un proceso normal de carga donde se puede ver que valor toma cada campo, etc.

3.1. Lógica Entre Mensajes JSON.

Según el protocolo OCPP, cada mensaje JSON tiene un formato definido. Y si se envía un mensaje **request** (.req), siempre se espera una respuesta **response** (.conf). Además el protocolo, también define una lógica de procedimiento, entre el primer mensaje JSON, su respuesta respectiva y el siguiente mensaje JSON. Como se verá brevemente en los esquemas planteados en esta sección.

3.1.1. Proceso de Encendido del SAVE.

Cuando se enciende el SAVE (luego de un reinicio o apagado), el primer mensaje a enviar al servidor será el BootNotification.req, para esperar confirmación de si el SAVE en cuestión, está habilitado, pendiente o rechazado para el uso en ese momento.

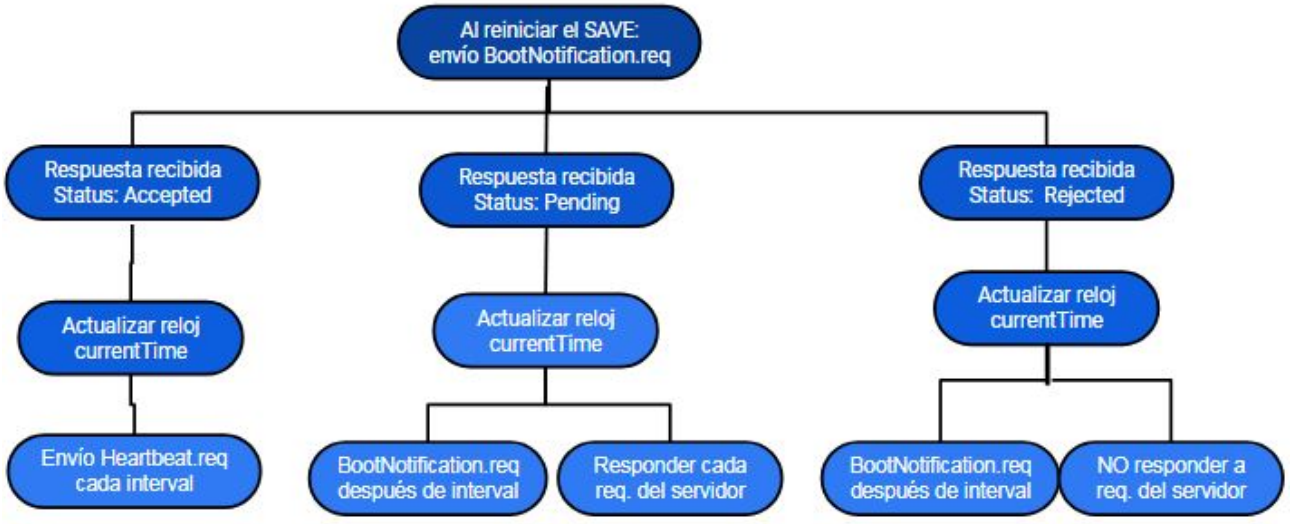


Figura 3.1: Lógica de encendido del SAVE.

Como se puede observar en la figura 3.1, dependiendo del valor del campo ***status***, el comportamiento será diferente:

- Si es *Accepted* el siguiente mensaje enviado por el SAVE será un Heartbeat.req entre lapsos de tiempo igual al valor ***interval***.
- Si es *Pending* el SAVE sólo responderá (.conf) a mensajes JSON comenzados (.req) desde el servidor, por un lapso de tiempo igual al valor ***interval***. Después de pasado ese tiempo, el SAVE enviará un nuevo BootNotification.req.

- Si es *Rejected* el SAVE no va a enviar ningún mensaje JSON, por un lapso de tiempo igual al valor ***interval***. Después de pasado ese tiempo, el SAVE enviará un nuevo BootNotification.req.

Y sin importar el valor del campo ***status*** de respuesta (BootNotification.conf), se recomienda que el SAVE actualice su reloj, con el valor dado por la respuesta en el campo ***currentTime***.

3.1.2. Inicio (o Parada) de una Sesión de Carga de VE.

Antes de iniciar una carga de un VE, el usuario deberá pasar su tarjeta por el lector del SAVE. El ***IdTag*** (ver sección A.3) obtenido a través del lector, lo enviará el SAVE en el mensaje JSON Authorize.req para solicitar al servidor que confirme, si el usuario que pide la carga, está habilitado o no para hacerlo.

Si el ***status*** (ver sección A.3) es aceptado, se habilita la sesión de carga. Pero si el ***status*** tiene un valor distinto, entonces se le muestra al usuario la razón del rechazo. Cuando el usuario quiera detener la carga, deberá pasar nuevamente su tarjeta por el lector del SAVE, y el proceso se repite, como muestra la figura 3.2.

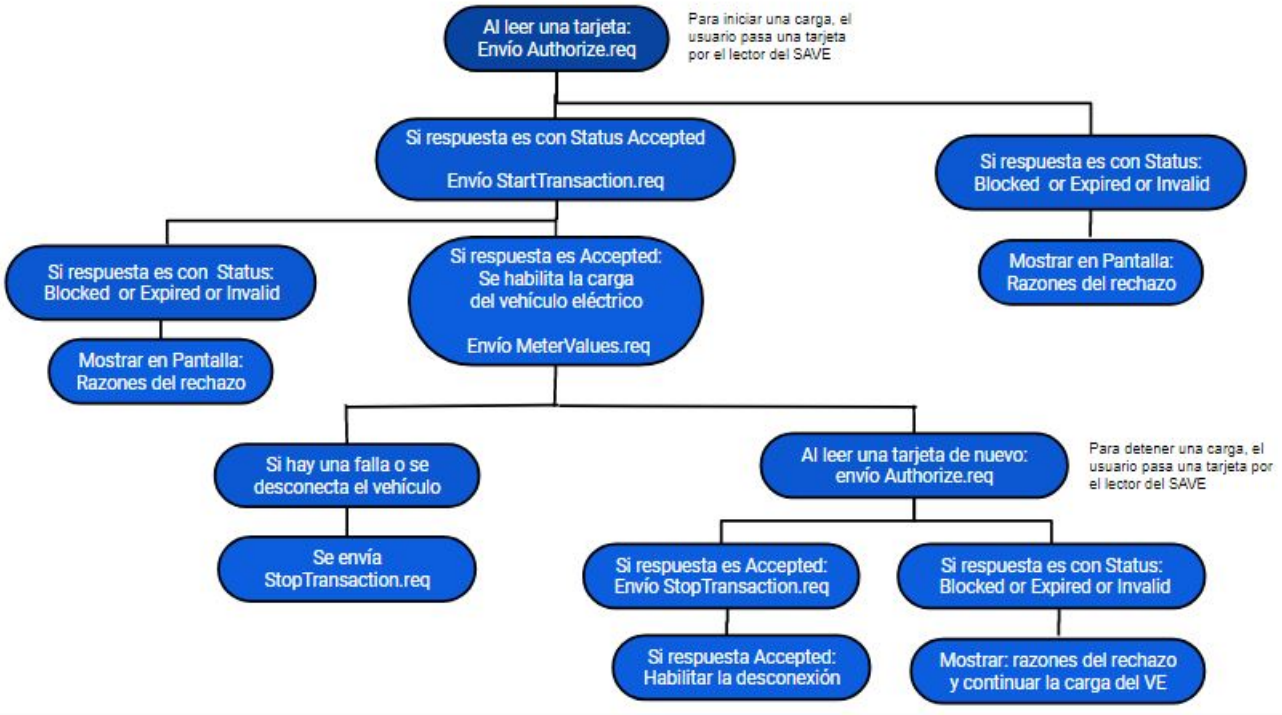


Figura 3.2: Lógica inicio (o parada) de una sesión de carga del SAVE.

3.2. Descripción de la Estructura de los Mensajes JSON.

La conexión que se realiza entre el servidor (en este caso, servidor de UTE) y el SAVE, es a través de WebSocket. Un WebSocket es una conexión full-duplex, es decir, se comporta como un canal donde los datos pueden tanto entrar o salir sin una clara relación entre ellos. El protocolo WebSocket por si solo no proporciona ninguna manera de relacionar los mensajes como solicitudes o respuestas.

Para codificar estas relaciones de petición/respuesta es necesario un protocolo de comunicación para que los mensajes sigan cierta estructura para poder ser enviados, leídos y relacionados entre si. Este problema en la comunicación aparece cada vez que se pretende utilizar WebSocket, por lo tanto, hay varios esquemas existentes para resolverlo.

Uno de los más utilizados es el llamado WAMP (ver [6]). Dado que el marco de la comunicación que se necesita entre el cargador y el servidor es simple (en comparación con otras aplicaciones donde se utiliza WebSocket) se define un marco propio de comunicación, inspirado en WAMP, dejando de lado lo que no es necesario y añadiendo lo que hace falta.

En la comunicación que se lleva a cabo entre el SAVE y el servidor es necesario diferenciar los mensajes en tres grandes grupos. La comunicación se da en primera instancia con un mensaje enviado (CALL) al "canal" (canal de comunicación bidireccional entre el SAVE y el centro de control de carga) el cual puede recibir una respuesta (CALLRESULT) o una explicación de por qué el mensaje no puede ser gestionado correctamente (CALLERROR). Por lo tanto, dentro de la estructura de los mensajes que se envían al "canal" se debe incluir si se trata de un mensaje de tipo CALL, CALLRESULT o CALLERROR.

Para hacer esta diferenciación entre los mensajes que se envían al "canal" y poder determinar cual corresponde a cada tipo se realiza la codificación que se muestra en la tabla 3.1.

Tipo de mensaje	MessageTypeID	Dirección
CALL	2	De cliente a servidor
CALLRESULT	3	De servidor a cliente
CALLERROR	4	De servidor a cliente

Tabla 3.1: Codificación para los mensajes JSON según su tipo.

Si un mensaje con un numero de "MenssageTypeId" que no corresponda a los valores que se muestran en la tabla 3.1 es enviado al "canal", al no entrar en ninguna de las categorías de CALL, CALLRESULT o CALLERROR no será tenido en cuenta y no se intentará leer la información contenida en él. Es decir, es un mensaje que nadie espera recibir, por lo tanto, nadie lo leerá.

El "MenssageTypeId" es una primer manera de diferenciar los mensajes entre si, pero no es la única que aparece en la estructura de los mensajes. Es en este punto donde aparece el "UniqueId" el cual sirve para identificar cada petición. El "UniqueId" es un valor que aparece en un mensaje de petición o llamada (CALL) el cual es diferente a todos los "UniqueId" previamente utilizados por el mismo remitente para los mensajes de llamada en la misma conexión WebSocket. El valor del "UniqueId" de los mensajes CALLRESULT o CALLERROR debe de ser igual al "UniqueId" del mensaje CALL al cual hacen referencia. El valor de "UniqueId" puede contener

tanto números como letras (mayúsculas y minúsculas) combinadas de la forma que se desee, con la única restricción de que no debe superar un máximo de 36 caracteres.

Resumiendo lo visto hasta ahora, cuando se envía en mensaje del tipo CALL (no importa si es del lado del SAVE o del servidor) éste tiene en su información el valor 2 en el campo correspondiente a "MenssageTypeId" y se le asigna un valor "UniqueId" único. La respuesta a este mensaje (que puede ser del lado del servidor o del SAVE) va a tener el valor 3 o 4 según sea un CALLRESULT o CALLERROR en el campo correspondiente a "MenssageTypeId" y el mismo valor de "UniqueId" correspondiente al mensaje CALL.

A continuación se realizará el análisis de la estructura de un mensaje de tipo CALL. Un mensaje de este tipo siempre consta de 4 elementos: Los elementos estándares "MenssageTypeId" y "UniqueId", una acción específica que se requiere en el otro lado y un contenido o "carga" útil (los argumentos de la acción). La sintaxis de un mensaje de tipo CALL se puede ver a continuación.

```
[
    MenssageTypeId,
    "UniqueId",
    "Action",
    {
        Payload
    }
]
```

Los campos de "MenssageTypeId" y de "UniqueId" ya se han explicado en párrafos anteriores. El campo denominado como "Action", como su nombre lo indica, hace referencia a la acción que esta destinada llevar a cabo dicho mensaje. En este campo llamado "Action" aparecen , por ejemplo, las acciones como "BootNotidication", "Authorize", "Heartbeat", "StartTransaction", etc.

El campo que queda analizar es "Payload" el cual contiene los argumentos pertinentes a la acción que se desea llevar a cabo (estos argumentos se verán con más detalle cuando se analice cada mensaje). Si el mensaje que se envía como CALL no necesita ningún argumento (esto sucede, por ejemplo, al enviar un "Heartbeat") el campo "Payload" debe de enviarse como un objeto vacío.

Para poder formar una idea más clara de esta estructura de mensaje se muestra a continuación un ejemplo de un mensaje de tipo CALL donde la acción que se desea llevar a acabo es "BootNotification".

```
[
    2,
    "19223201",
    "BootNotification",
    {
        "ChargePointVendor":"VendorX",
        "chargePointModel":"SingleSocketCharger",
    }
]
```

Si el mensaje CALL enviado se puede llevar a cabo correctamente el resultado será una respuesta de tipo CALLRESULT. Esto no quiere decir que la respuesta sea positiva, solo significa que la petición se llevo a cabo correctamente.

Un CALLRESULT siempre consta de 3 elementos:

Los dos elementos estándar “MenssageTypeId” y “UniqueId” y un campo para contenido o “carga” útil, que contiene una respuesta a la acción del mensaje CALL. La sintaxis de un mensaje CALLRESULT se muestra a continuación.

```
[
    MenssageTypeId,
    "UniqueId",
    {
        Payload
    }
]
```

Los campos que aparecen en la sintaxis de un mensaje CALLRESULT ya son conocidos, por lo tanto, se verá directamente un ejemplo de un CALLRESULT correspondiente a un BootNotification. Esto se puede ver en la siguiente estructura.

```
[
    3,
    "19223201",
    {
        "status":"Accepted",
        "currentTime":"2020-06-03T20:53:32.486Z",
        "interval":300,
    }
]
```

Notar que el primer valor que aparece en el mensaje de la estructura CALL es un 2, lo que da la información que es un mensaje de tipo CALL con el “UniqueId” asociado (en este ejemplo 19223201). Por otra parte, se ve en la estructura anterior que el primer valor que aparece es un 3, por lo tanto, corresponde a un mensaje de tipo CALLRESULT y con el valor de “UniqueId” (que es el mismo al del mensaje CALL) se puede saber a que mensaje CALL esta vinculada ésta respuesta. Esta asociación de los “UniqueId” cobra relevante sentido cuando en cierto momento de la comunicación entre el SAVE y el servidor, hay más de un mensaje de tipo CALL en el ”canal” de comunicación para los cuales se esta a la espera de una respuesta.

Solo resta ver el caso de un mensaje de tipo CALLERROR. Este tipo de mensaje viene en respuesta de un mensaje de tipo CALL pero si se dan las siguientes condiciones:

1. Si durante el envío del mensaje ocurre un error. Esto puede ser provocado por problemas en la red.

2. Se recibe bien el mensaje de tipo CALL pero el contenido del mensaje no cumple con los requisitos de un mensaje adecuado. Es decir que pueden faltar campos que se consideran obligatorios en la comunicación, que el mensaje CALL este usando un “UniqueId” que ya haya sido utilizado o se este usando en ese momento, etc.

Un mensaje de tipo CALLERROR consta de 5 elementos principales: los elementos estándar "MenssageTypeId" y "UniqueId", dos campos llamados "errorCode" y "errorDescription" y un campo de tipo objeto llamado "errorDetails". La sintaxis del mensaje CALLERROR se puede ver a continuación.

```
[
    MenssageTypeId,
    "UniqueId",
    "errorCode",
    "errorDescription",
    {
        errorDetails
    }
]
```

Los dos primeros campos cumplen la misma función que ya se ha visto en los mensajes anteriores. El campo "errorCode" se utiliza para asociar el problema ocurrido a errores ya tabulados. El campo "errorDescription" se debe de completar (si se cree necesario) con la descripción del error que ha ocurrido. El campo "errorDetails" se utiliza (si se cree necesario) para dar detalles sobre el error ocurrido. Estos dos últimos campos se utilizan y completan a criterio del emisario del mensaje CALLERROR con el nivel de detalle e información que se crea necesario. El campo "errorCode" es un campo que obligatoriamente debe de ser completado siguiendo los criterios de la tabla 4.1.

ErrorCode	Descripción
NotSupported	Se reconoció la acción solicitada pero no es soportada por el receptor.
InternalError	Se ha producido un error interno y el receptor no pudo procesar la acción solicitada con éxito.
ProtocolError	El contenido del mensaje o la acción solicitada no cumple con el protocolo.
SecurityError	Durante la tramitación de la acción se produjo un problema de seguridad impidiendo al receptor completar con éxito la acción.
FormationViolation	El contenido del campo "Payload" es sintácticamente incorrecto o no se ajusta al pedido de acción solicitado.
PropertyConstraintViolation	El contenido del campo "Payload" es sintácticamente correcto pero al menos un campo contiene un valor no válido.
OccurenceConstraintViolation	El contenido del campo "Payload" es sintácticamente correcto pero al menos uno de los campos viola los valores permitidos.
TypeConstraintViolation	El contenido del campo "Payload" es sintácticamente correcto pero al menos uno de los campos viola restricciones de tipo de datos.
GenericError	Cualquier otro error no cubierto por los anteriores.

Tabla 3.2: Codificación para los códigos de error.

Como se puede ver en la tabla 4.1 hay distintos tipos de errores tabulados los cuales se utilizan según corresponda.

En esta sección quedó presentada la sintaxis estructural y campos más comunes que se utilizan en los mensajes presentes en la comunicación entre el centro de control de carga y el SAVE. En las secciones futuras se verán y analizarán con más detalle estos mensajes ahondando en cada uno de los campos según corresponda.

3.3. Resumen de Mensajes OCPP 1.6 Seleccionados.

Mas allá de que hay una gran variedad de mensajes posibles para la comunicación entre el SAVE y el servidor, no quiere decir que todos ellos sean estrictamente necesarios para una correcta comunicación o para concretar un proceso de carga de forma correcta. Para la programación del SAVE se decidió implementar los mensajes que sí son estrictamente necesarios para la carga de un vehículo y que a su vez son los únicos mensajes que el servidor, como centro de control de carga, espera recibir y por ende tiene implementada una respuesta. Es decir, el resto de los mensajes en su gran mayoría tampoco están implementados en el servidor (de UTE), por lo tanto, si el SAVE envía alguno de los mensajes que el servidor no tiene implementado, no recibiría ninguna respuesta.

Esto no quiere decir que los mensajes que no son imprescindibles para un proceso normal de carga no aporten a la comunicación o a un mejor servicio para los usuarios. Por ejemplo, si un usuario quiere cargar su VE y asegurarse que va a poder hacerlo (que no haya otro usuario utilizando el SAVE) podría comunicarse con el servidor y éste a través del mensaje "ReserveNow" podaría comunicarle al SAVE en qué horario solo puede cargar su VE el usuario que realizó la reserva. La estructura de este menaje como del resto de los mensajes implementados en este caso se pueden ver en el anexo (ver Anexo A.2).

Los mensajes seleccionados, son los suficientes para implementar una sesión de carga convencional. Cabe destacar que en este caso, cada mensaje de **request** (.req) es enviado siempre por el SAVE y el que devuelve un mensaje **response** (.conf) es siempre el servidor.

Como se explicó anteriormente, la comunicación WebSocket permite que sea el servidor quien inicie la comunicación, con un mensaje de **request**, pero en los mensajes implementados no hay ninguno que aplique esa lógica. Entonces, si el servidor envía un mensaje de **request**, el SAVE implementado no tiene ningún mensaje **response** (.conf) correspondiente para enviar.

1) Authorize.

Antes que el usuario del VE pueda iniciar o detener una sesión de carga, el SAVE tiene que autorizar dicha operación. Es obligatorio que el SAVE suministre la energía, solo después de la autorización por parte del servidor. Al detener una sesión de carga, es recomendable que el SAVE solo envíe un Authorize.req, cuando el identificador utilizado para detener la transacción es diferente del identificador que inició la transacción.

Al recibir un mensaje Authorize.req, el servidor tiene la obligación de responder con un mensaje Authorize.conf. Esta respuesta indica si el servidor acepta o no el idTag. Si el servidor acepta el idTag, entonces la PDU de respuesta es opcional que incluya un campo parentIdTag, pero es obligatorio que incluya un valor de estado de autorización que indique la aceptación o una razón para el rechazo.

2) BootNotification.

Después de un reinicio, es recomendable que el SAVE envíe un mensaje al servidor con la información de su configuración. Y es obligatorio que el servidor responda, indicando si acepta o no la puesta en marcha de ese SAVE en ese instante. Mientras no haya respuesta del servidor, el SAVE no debería enviar ningún otro mensaje al servidor.

- Si el servidor responde **Accepted**: El SAVE configurará un envío periódico, del mensaje Heartbeat, cada vez que se cumpla un período de tiempo establecido por la respuesta del servidor, en el campo **interval**.
- Si el servidor responde **Rejected**: El SAVE no deberá enviar ningún mensaje al servidor, hasta que el tiempo de espera (determinado en **interval**) se haya agotado. Luego se reenvía un nuevo BootNotification.req.
- Si el servidor responde **Pending**: El SAVE sólo deberá responder a mensajes PDU iniciados desde el servidor, hasta que el tiempo de espera (determinado en **interval**) se haya agotado. Luego se reenvía un nuevo BootNotification.req.

3) Heartbeat.

Para que el servidor sepa que un SAVE todavía está conectado, dicho SAVE envía un Heartbeat.req después de un intervalo de tiempo configurable. Al recibir un Heartbeat.req el servidor esta obligado a responder con un Heartbeat.conf. (Con JSON mediante WebSocket, el envío de Heartbeat no es obligatorio.)

4) MeterValues.

Un SAVE puede tomar muestras del medidor de energía u otro hardware (sensor/transductor) para proporcionar información sobre los valores medidos. Depende del SAVE decidir cuándo enviará los valores del medidor mediante el mensaje MeterValues.req
La PDU enviado por el SAVE deberá contener para cada muestra:

(1). La identificación del conector del cual se tomaron las muestras. Si el ID de conector es 0, la medida está asociada con todo el SAVE.

(2). El ID de la transacción con la que están relacionados estos valores, si corresponde. Si no hay ninguna transacción en curso o si los valores se toman del medidor principal, se puede omitir la identificación de la transacción.

(3). Uno o más elementos meterValue, de tipo MeterValue, cada uno representa un conjunto de uno o más valores de datos tomados en un momento determinado en el tiempo.

Cada elemento MeterValue contiene una marca de tiempo y un conjunto de uno o más elementos de valores muestreados, todos capturados en el mismo instante. Cada elemento muestreado contiene un dato de valor único. La naturaleza de cada valor muestreado está determinado por los campos opcionales de medición, contexto, ubicación, unidad, fase y formato.

Al recibir una PDU MeterValues.req, el servidor está obligado a responder con un MeterValues.conf. Es probable que el servidor aplique controles de validez a los datos contenidos en el MeterValues.req que recibió. El resultado de tales comprobaciones, no debe causar que el servidor deje de responder con un MeterValues.conf. No responder con un MeterValues.conf solo hará que el SAVE intente enviar nuevamente el mismo mensaje.

5) StartTransaction.

El SAVE debe enviar una PDU StartTransaction.req al servidor, para informar que se ha iniciado una sesión de carga. Si esta sesión de carga finaliza, el StartTransaction.req debe contener el ID de correspondiente a la misma.

Al recibir una PDU StartTransaction.req, el servidor debería responder con una PDU StartTransaction.conf. Esta PDU de respuesta es obligatorio que incluya un ID de transacción y un valor de estado de autorización.

Es obligatorio que el servidor verifique la validez del idTag del StartTransaction.req que recibió y es probable que verifique validez de otros datos también. El resultado de tales comprobaciones, no debe causar que el servidor deje de responder con un StartTransaction.conf. No responder con un StartTransaction.req solo hará que el SAVE intente enviar nuevamente el mismo mensaje.

6) StopTransaction.

El SAVE debe enviar una PDU StopTransaction.req al servidor, para informar que se ha detenido una sesión de carga. El SAVE puede detener una sesión de carga cuando el cable es desconectado del VE. Esta funcionalidad evita el sabotaje, porque detiene el suministro de energía, evitando que un usuario desconecte el cable de un VE y lo conecte a otro, utilizando la misma sesión de carga.

Una PDU StopTransaction.req puede contener un elemento opcional TransactionData para proporcionar más detalles al servidor. El elemento opcional TransactionData es un contenedor para cualquier número de MeterValues, que utiliza la misma estructura de datos que los elementos meterValue de la PDU MeterValues.req. Mientras que el idTag en la PDU StopTransacrion.req puede ser omitido cuando el SAVE precise detener la sesión de carga por si mismo, por ejemplo cuando el SAVE precisa resetearse.

Al recibir una PDU StopTransaction.req, el servidor está obligado a responder con una PDU StopTransaction.conf. El servidor no puede evitar que una sesión de carga se detenga. Lo que puede hacer, es informar al SAVE que recibió la PDU StopTransaction.req y le envíe información acerca del idTag utilizado para detener la sesión de carga. Esta información la tendrá que actualizar el SAVE en caso de tener implementado un Authorization Caché.

Es probable que el servidor aplique controles de validez a los datos contenidos en el StopTransaction.req que recibió. El resultado de tales comprobaciones, no debe causar que el servidor deje de responder con un StopTransaction.conf. No responder con un StopTransaction.conf solo hará que el SAVE intente enviar nuevamente el mismo mensaje.

3.4. Descripción de la Comunicación SAVE-Centro de Control de Carga en un Proceso Normal de Carga.

En esta sección se describirá la comunicación llevada a cabo desde el momento que el SAVE es energizado, entra en funcionamiento y realiza una carga de un VE de forma satisfactoria.

El análisis del proceso comienza en el momento que el SAVE es energizado. En ese momento tanto la Raspberry como el Arduino (que son componentes vitales del SAVE) (en secciones posteriores se verán en detalle estos componentes) comienzan su funcionamiento. El Arduino por defecto comienza a correr el programa que lleva guardado (el cual se puede ver en el anexo en la sección C.1) y a su vez la Raspberry fue configurada de forma tal que comienza a correr el programa llamado SAVE.js (el cual se puede ver en el anexo en la sección B.1). Una vez que estos programas comienzan a correr se podría decir que el "Cerebro" del SAVE ha despertado, dando lugar a la comunicación con el centro de control de carga. En esta sección no se entrará en detalle de cómo se realiza el envío, la recepción o el análisis de los mensajes o de otros datos que son censados para realizar la comunicación.

Una vez que el SAVE esta energizado y con una correcta apertura del canal de comunicación con el servidor, lo primero que hace es enviar el mensaje "BootNotification" al servidor. A continuación se muestra cómo es el mensaje enviado.

```
[
    2,
    "BBS5648",
    "BootNotification",
    {
        "chargePointModel":"FinDeCarrera",
        "chargePointVendor":"GutierrezHaltyMango"
    }
]
```

En este caso se puede ver que valores toman los campos "chargePointModel" y "charge-PointVendor" los cuales son chequeados por el servidor para corroborar que pertenecen a un SAVE perteneciente a su sistema. Luego que el SAVE envía el mensaje recibe la respuesta del servidor. Esta respuesta se puede ver a continuación.

```
[
    3,
    "BBS5648",
    {
        "status":"Accepted",
        "currentTime":"2020-06-03T20:53:32.486Z",
        "interval":300,
    }
]
```

Como se puede ver en los mensajes tanto enviado como recibido, los ID de los mensajes son los mismos (parámetro que marca que uno es la respuesta del otro). Recordando que se está analizando el caso donde la comunicación transcurre con normalidad se puede ver que el campo

"status" que aparece en el mensaje de repuesta es "Accepted". Al estar el SAVE dentro de la condición de "Aceptado", se toma el valor del campo "interval" (en este caso vale 300 ms) y se utiliza ese valor para enviar cada intervalos de tiempo T (con $T =$ "$interval''$) el mensaje "Heartbeat". Este mensaje que será enviado de forma periódica se puede ver a continuación.

```
[
    2,
    "45dasf4CVD45MJGN694",
    "Heartbeat",
    {
    }
]
```

Cada vez que el SAVE envía el mensaje "Hertbeat" el servidor envía como respuesta el siguiente mensaje.

```
[
    3,
    "45dasf4CVD45MJGN694",
    {
    "currentTime":"2020-06-03T20:54:12.486Z",
    }
]
```

Una vez que se llegó a este estado, el SAVE permanece activo y espera la llegada de un usuario. En todo momento se le informa al usuario por medio de la pantalla que pase su tarjeta por el lector. Luego de que la tarjeta es leída por el SAVE se envía al servidor el mensaje "Authorize" donde en el campo "idTag" se ubica la información de la tarjeta. El mensaje Authorize enviado por el SAVE se puede ver a continuación.

```
[
    2,
    "NASsdC164ACSASC",
    "Authorize",
    {
        "idTag":"8BC57123"
    }
]
```

Luego de que el SAVE envía el mensaje, espera la respuesta del servidor. La respuesta del servidor en este caso es la siguiente.

```
[
    3,
    "NASsdC164ACSASC",
    {
    idTagInfo:
        {
        "status":"Accepted"
        }
    }
]
```

Como se puede ver en el mensaje el usuario es aceptado, es decir, que es un usuario válido para el servidor, por lo tanto el SAVE por intermedio de la pantalla le avisa al usuario que conecte el VE (en caso de no haberlo conectado antes).

Una vez que el VE es conectado el SAVE chequea si el cable que se utilizó es adecuado para una carga. Si el cable cumple con las condiciones para soportar una carga, tanto el SAVE como el VE están en condiciones de iniciar la carga y el usuario esta autorizado, por lo tanto, se envía el mensaje "StartTransaction" el cual se muestra a continuación.

```
[
  2,
  "UnsSFFSd1SFasd1565",
  "StartTransaction",
  {
    "connectorId": 1,
    "idTag":"8BC57123",
    "meterStart": 8508,
    "timestamp":"2020-06-03T20:58:44.486Z"
  }
]
```

Una vez que el SAVE envía la petición para comenzar una transacción, espera la respuesta del servidor, siendo ésta la siguiente.

```
[
    3,
    "UnsSFFSd1SFasd1565",
    "transactionId": 1797,
    {
        "idTagInfo",
        {
        "status":"Accepted"
        }
    }
]
```

Una vez que la transacción es autorizada, el SAVE le envía al VE la información de que corriente máxima puede utilizar para la carga. Esto lo hace enviando la señal PWM, y cierra

el contactor para permitir el flujo de energía. Una vez hecho esto, el VE comienza a cargarse. Todo esto será explicado con detalle en secciones posteriores.

Por otro lado, el SAVE comienza a enviar el mensaje "MeterValues" para informar la energía que el VE va consumiendo. Este mensaje se puede ver a continuación.


```
[
    2,
    "58NVUT61ca92csaSASC",
    "MeterValues",
    {
    "connectorId": ,
    "meterValue":
    [
        {
        "sampledValue":
        [
            {
            "measurand":"Energy.Active.Import.Register",
            "unit":"Wh",
            "value":"7023"
            }
        ],
        "timestamp":"2020-06-03T20:59:10.486Z"
        }
    ]
  }
]
```


De este mensaje se obtiene la siguiente respuesta por parte del servidor.


```
[
    3,
    "58NVUT61ca92csaSASC",
    {
    }
]
```


El envió del mensaje "MeterValues" de forma periódica ocurre mientras la sesión de carga esta abierta. Una sesión de carga se termina si el usuario desea terminarla (pasando la tarjeta por el lector), si se desconecta el vehículo, o si ocurre algún otro problema y la carga se ve interrumpida. En este caso la sesión de carga es detenida por el usuario, por lo tanto se envía el mensaje "StopTransaction" que se puede ver a continuación.


```
[
  2,
  "NCOS48Spo",
  "StopTransaction",
```

```
  {
    "transactionId": 1797,
    "timestamp":"2020-06-03T21:24:06.486Z",
    "meterStop": 8580
  }
]
```

La respuesta al “StopTransaction” enviada por el servidor es la siguiente.

```
[
    3,
    "NCOS48Spo",
    {
    "idTagInfo",
        {
        "status":"Accepted",
        }
    }
]
```

Como se puede ver, es aceptada la petición a cerrar la sesión de carga, por lo tanto el SAVE le “avisa” al VE que deje de consumir energía (lo hace quitando la señal PWM) e inmediatamente abre el contactor. Una vez echo esto, el SAVE deja de enviar mensajes “MeterValues” y le informa al usuario por intermedio de la pantalla que la carga ha finalizado y que puede desconectar el VE.

Una vez que el VE se desconecta el SAVE vuelve a su estado inicial de esperar un nuevo usuario para comenzar una nueva sesión de carga.

CAPÍTULO 4

CÓDIGO IMPLEMENTADO SEGÚN PROTOCOLO OCPP.

4.1. Descripción del Código Implementado en Raspberry.

El código a analizar en esta sección, ejecuta la comunicación entre el SAVE y el servidor de UTE. Como dicha comunicación tiene una cantidad finita de mensajes a enviar o recibir, el código fue implementado según una lógica de estados, en la que en **cada estado: envía o recibe un mensaje**.

Estado	Descripción	Estado siguiente
E0	Envío del **BootNotification Request**.	**E1**
E1	Lectura **BootNotification Response** y seteo del envío periódico del **Heartbeat**.	**E2** - E1 - E8
E2	Envío del **Authorize Request** y control del error temporal del Arduino.	**E3** - **E9**
E3	Lectura **Authorize Response**.	**E4** - E2 - E0
E4	Envío del **StartTransaction Request**.	**E5** - E2
E5	Lectura **StartTransaction Response** y seteo del envío periódico del **MeterValue**.	**E6** - E4 - E2 - E0
E6	Envío del **StopTransaction Request**, validando tarjeta antes.	**E7** - E8
E7	Lectura **StopTransaction Response** y detener envío del **MeterValue** y del **Heartbeat**.	**E2** - E8 - E6
E8	Envío **StopTransaction Request**, sin validar tarjeta.	E7
E9	Error definitivo en Arduino - Esperar recuperación del mismo.	E0

Tabla 4.1: Resumen de lo realizado en cada estado.

El código implementado para cumplir con lo mencionado, se presenta en el Anexo B. Y los archivos correspondientes se pueden encontrar en el soporte digital que acompaña este documento en la carpeta Raspberry PI.

- En el estado **E0** el SAVE está fuera de servicio. La Raspberry envía un mensaje de solicitud **BootNotification.req** al servidor y luego pasa al estado **E1** para leer la respuesta. Al estado **E0** se llega siempre que se necesite reiniciar la comunicación con el servidor.
- En el estado **E1** la Raspberry espera recibir la respuesta **BootNotification.conf** del servidor. Sólo si el contenido de la respuesta en el **Status** es **Accepted**, entonces el SAVE queda habilitado para ser usado y la Raspberry pasa al estado **E2**.
- La Raspberry permanecerá en el estado **E2**, hasta que se haya leído una tarjeta. Cuando la Raspberry recibe[1] el código identificador (**IdTag**) de la tarjeta leída, lo guarda en un mensaje de solicitud **Authorize.req** y lo envía al servidor para validar la tarjeta.
- En el estado **E3** la Raspberry espera recibir la respuesta **Authorize.conf** del servidor. Sólo si el contenido de la respuesta en el **Status** es **Accepted**, entonces se pasa al **E4**.
- En el estado **E4** se envía una solicitud **StartTransaction.req** al servidor, para poder iniciar la sesión de carga de un VE, junto con el código de la tarjeta válida (**IdTag**) asociado a dicha solicitud.
- En el estado **E5** la Raspberry espera recibir la respuesta **StartTransaction.conf** del servidor. Sólo si el contenido de la respuesta en el **Status** es **Accepted**, entonces la Raspberry notifica al Arduino que inició la sesión de carga[2] y luego pasa al estado **E6**.
- La Raspberry permanecerá en el estado **E6**, hasta que se lea la misma tarjeta válida[3] o hasta que el conector del VE se desconecte intempestivamente del SAVE. En dicho caso, se envía una solicitud **StopTransaction.req** al servidor, para detener la sesión de carga.
- En el estado **E7** la Raspberry espera recibir la respuesta **StopTransaction.conf** del servidor. Sólo si el contenido de la respuesta en el **Status** es **Accepted**, entonces la Raspberry notifica al Arduino que se detiene la sesión de carga y luego pasa al **E2** a esperar una tarjeta válida para iniciar una nueva sesión.
- En el estado **E8** se envía una solicitud **StopTransaction.req** al servidor, para detener la sesión de carga. Es análogo al estado **E6** pero sin esperar que se lea una tarjeta o que se desconecte el conector del VE. Se llega a este estado en caso de querer cerrar una sesión de carga que tuvo error de comunicación con el servidor o un error de conexión con el VE.
- En el estado **E9** se permanece siempre que se detecte un error en la conexión con el VE.

En el diagrama de estados de la figura 4.1 se representa de forma gráfica las transiciones de estado y en la tabla 4.2 se exponen las condiciones requeridas para realizar cada una de las transiciones. Las transiciones consideradas exitosas son marcadas en color verde, en la figura 4.1, para mayor comprensión. Y una secuencia exitosa tendría el siguiente orden de transición: **E0** → **E1** → **E2** → **E3** → **E4** → **E5** → **E6** → **E7** → **E2**

[1] Es el Arduino el que envía el código identificador de la tarjeta a la Raspberry. Y obtiene dicho código mediante un módulo lector de tarjetas (RFID RC522) compatible con Arduino.

[2] Cuando se notifica al Arduino del inicio de sesión de carga, este verifica si el VE está conectado al SAVE y luego envía una señal PWM con un Duty Cycle correspondiente a la máxima corriente admisible del sistema.

[3] Si el código de tarjeta leído por el Arduino, es distinto al código de la tarjeta que inició la sesión de carga, entonces la Raspberry le notifica al Arduino que esa sesión no se va a detener porque se rechazó la tarjeta.

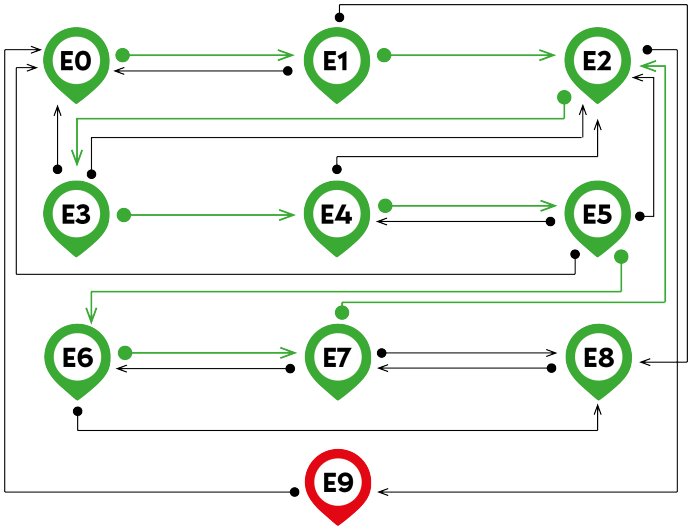


Figura 4.1: Diagrama de estados en la Raspberry

Inicial	Final	Condición de transición
E0	**E1**	Se envía BootNotification.req y se pasa al estado E1, para esperar la respuesta.
E1	**E2**	Si la respuesta es "Accepted" y la transacción anterior está cerrada, se pasa a E2.
E1	E0	Si la respuesta NO es "Accepted", entonces se vuelve a E0. Si no hay respuesta luego de un tiempo (time_response_boot), entonces vuelve al E0.
E1	E8	Si la respuesta es "Accepted", pero hay una transacción sin cerrar, se pasa al E8.
E2	**E3**	Al recibir un código de tarjeta, se envía un Authorize.req, y se pasa al E3.
E2	**E9**	Si recibe un código error del Arduino, se va al E9, para esperar el restablecimiento.
E3	**E4**	Si la respuesta Authorize.conf del servidor, es "Accepted", se pasa al E4.
E3	E2	Si la respuesta NO es "Accepted', entonces vuelve a E2. Si no hay respuesta luego de un tiempo (time_response_authorize), se vuelve al E2. Si se recibe un código error del Arduino, se vuelve a E2 a verificar si es transitorio.
E3	E0	Si no hay respuesta luego de un tiempo (time_reset_authorize), se vuelve al inicio E0.
E4	**E5**	Se envía StartTransaction.req y se pasa al estado E5, para esperar la respuesta.
E4	E2	Si se recibe un código error del Arduino, vuelve a E2 a verificar si es transitorio.
E5	**E6**	Si la respuesta StartTransaction.conf del servidor, es "Accepted", se pasa al E6
E5	E4	Si no hay respuesta en un tiempo (time_response_startTransaction), se vuelve al E4.
E5	E2	Si la respuesta NO es "Accepted', entonces vuelve a E2, para esperar nueva tarjeta
E5	E0	Si no hay respuesta en un tiempo (time_reset_startTransaction), vuelve al inicio E0.
E6	**E7**	Si al recibir un código de tarjeta, es el mismo código que el leído antes, se va al E7. Si Arduino notifica que el vehículo se desconecto del SAVE, entonces se va al E7.
E6	E8	Si se recibe código error del Arduino, se va al estado E8.
E7	**E2**	Si la respuesta StopTransaction.conf del servidor, es "Accepted", se pasa al E2.
E7	E6	Si la respuesta NO es "Accepted', entonces se vuelve a E6 a esperar otra tarjeta.
E7	E8	Si no hay respuesta luego de un tiempo (time_response_stopTransaction), se va al E8.
E8	E7	Siempre se envía StopTransaction.req, sin esperar tarjeta.
E9	E0	Si se recibe un código de restablecimiento del Arduino, se vuelve al inicio E0.

Tabla 4.2: Resumen de lo realizado en cada estado.

4.1.1. Función Principal *main().*

La función principal ***main()***, ejecuta un bucle (loop) que se repite en cada intervalo de tiempo "*time_main*". El bucle verifica el valor de la variable (***main_status***) y dependiendo del valor de dicha variable, se ejecutará solamente el código que corresponda a cierto estado. Por ejemplo, si (***main_status***) vale E5, sólo se ejecutará el código que corresponda al estado E5.

La función ***main()*** se ejecuta, a partir de una llamada desde la función ***connect()***. Es la función ***connect()*** la que verifica si el servidor está disponible para realizar la comunicación (ver sección 4.1.13.4).

4.1.2. Estado E0: Envío del BootNotification Request.

Se envía el mensaje **BootNotification** utilizando la función ***envio.mensaje(mens,socket,dato)***. Esta función será explicada en la sección 4.1.13.1.

Luego se resetea la variable **menssage_recived**, porque previo a ejecutar este estado, dicha variable puede que no esté vacía. También se le notifica al Arduino, que la Raspberry aún no está lista, enviando un número 0 como argumento de la función ***port.write()*** (ver 4.1.13.3)

Finalmente, se asigna a la variable (***main_status***) el valor E1, para que en el siguiente intervalo de tiempo "*time_main*" se ejecute el código del estado E1.

4.1.3. Estado E1: Lectura BootNotification Response y Seteo del Envío Periódico del Heartbeat.

Como fue explicado en la sección anterior, el servidor deberá responder, enviando un JSON con el mismo ID que tenía el último JSON enviado por la Raspberry. Además, en el caso del **BootNotification**, el servidor deberá responder si el Status es "*Accepted*" o no.

- **Si el mensaje recibido NO tiene el mismo ID que el BootNotification enviado.**

  Luego de un intervalo de tiempo "*time_response_boot*", se vuelve al estado anterior EO, para enviar un nuevo **BootNotification** al servidor.

  Para lograr dicho objetivo, se setea por única vez la función ***SetInterval()***, que va a ejecutar las líneas de código que contenga dicha función en su interior, cada vez que pase un periodo de tiempo "*time_response_boot*".

- **Si el mensaje recibido tiene el mismo ID que el BootNotification enviado.**

  Pero el Status NO es "*Accepted*".

  Luego de un intervalo de tiempo, igual al valor de **interval** de la respuesta recibida por el servidor, se vuelve al estado anterior EO, para enviar un nuevo **BootNotification** al servidor.

- **Si el mensaje recibido tiene el mismo ID que el BootNotification enviado.**

  Y el Status es "*Accepted*".

  Primero se le notifica al Arduino - enviando un 1 mediante la función ***port.write()*** - que la Raspberry está lista para recibir una nueva tarjeta. También tiene que resetear los

dos ***SetInterval()*** que se pudieron haber ejecutado, si cualquiera de los otros dos casos sucedió anteriormente.

Mediante la función ***SetInterval()*** se setea el envío periódico de un **Heartbeat**, en cada intervalo de tiempo igual al valor de **interval** de la respuesta recibida por el servidor. Y lo que se ejecutará, luego de transcurrido dicho intervalo, es el llamado de la función ***envio.mensaje(mens,socket)*** para envíar un **Heartbeat**, que su respuesta no será leída en ningún estado en particular, porque la respuesta no generará ningún cambio en el comportamiento del SAVE.

Finalmente, se realiza una ultima verificación: Si hubo una transacción anterior, que no fue cerrada, habrá que cerrarla antes de empezar una nueva.

- Si la **TransactionID** del último mensaje **StopTransaction** es cero, entonces se sabe que ya se cerró la ultima transacción, y se puede pasar al estado E2 para recibir una nueva tarjeta y enviar un mensaje **Authorize** al servidor.

- Pero en caso contrario, si **TransactionID** es distinta de cero, entonces hay una transacción anterior que sigue abierta. Por lo tanto, habrá que pasar al estado E8 para enviar un **StopTransaction**, sin esperar una nueva tarjeta, y así cerrar dicha transacción.

4.1.4. Estado E2: Envío del Authorize Request y Control del Error Temporal del Arduino.

Debido a que en esta estado, el Arduino ya sabe que que la Raspberry está lista para recibir una nueva tarjeta; es a partir de este estado en adelante, que el Arduino va a poder enviar códigos de tarjetas leídas o códigos de error, cuando haya un error de conexión con el vehículo.

Para la lectura de datos enviados por el Arduino, en la Raspberry se utiliza la función ***parser.on('data', function(data){arduino_data=data})*** (ver sección 4.1.13.3) que guarda en la variable ***arduino_data*** el dato leído. Esta función será explicada en detalle, luego de explicar los nueve estados.

En este estado, se verifica la variable ***arduino_data***:

- Si ***arduino_data*** es cero, entonces hay un error definitivo en el Arduino, por lo que hay que borrar el dato leído y pasar al estado E9 para esperar a que el Arduino se recomponga - el Arduino envía un 1, cuando se recompone -.

- Si ***arduino_data*** no es cero (ni tampoco es 1), entonces es seguro que el Arduino está enviando el código de una tarjeta leída. Así que, el siguiente paso es guardar dicho código en la variable ***tarjeta***, y resetear la variable ***arduino_data*** para la próxima lectura.

  Luego se envía el mensaje **Authorize** al servidor de UTE, utilizando la función auxiliar ***envio.mensaje(mens,socket,dato)***.
  En este caso los argumentos son: **mens** = "Autorizacion", **socket** es igual al socket utilizado en el programa principal y **dato** = ***tarjeta.slice(0,-1)***, que el mismo código guardado en ***tarjeta***, pero sin el carácter de fin de línea.

  Finalmente, se asigna a la variable (***main_status***) el valor E3, para que en el siguiente intervalo de tiempo "*time_main*" se ejecute el código del estado E3, para leer la respuesta del servidor al **Authorize** enviado.

4.1.5. Estado E3: Lectura Authorize Response.

El servidor deberá responder, enviando un JSON con el mismo ID que tenía el último JSON enviado por la Raspberry. Además, en el caso del **Authorize**, el servidor deberá responder si el Status es "*Accepted*" o no.

Mientras tanto, el Arduino puede enviar un código error, si hay error de conexión con el VE. Por eso, se verifica durante el estado E3, si ***arduino_data*** es cero o no.

En caso de que sea cero, entonces se considera un error temporal, por lo que se resetea la variable ***arduino_data*** (para luego poder leer otro dato) y pasar al estado E2. En el estado E2 se espera a que el Arduino se recomponga - el Arduino envía un 1- o que sea un error definitivo - el Arduino envía nuevamente un cero, unos 30 segundos después -.

- **Si el mensaje recibido NO tiene el mismo ID que el Authorize enviado.**

  Luego de un intervalo de tiempo "*time_response_authorize*", se vuelve al estado anterior E2, para enviar un nuevo **Authorize** al servidor.

  Para lograr dicho objetivo, se setea por única vez la función ***SetInterval()***, que va a ejecutar las líneas de código que contenga dicha función en su interior, cada vez que pase un periodo de tiempo "*time_response_authorize*".

  Al mismo tiempo, se setea por única vez, la función ***SetTimeout()***, para contar el pasaje del tiempo. Cuando se llegue al tiempo máximo, seteado en la variable "*time_reset_authorize*", entonces se ejecutará una sola vez, el código que contenga dicha función en su interior.

  En este caso, luego de un intervalo de tiempo "*time_reset_authorize*", se vuelve al estado inicial E0, para enviar un **BootNotification** al servidor y notificar al Arduino que la Raspberry no está lista. Y también se resetea el ***SetInterval()*** anterior, porque ya no va a ser necesario re-enviar el mensaje **Authorize**.

- **Si el mensaje recibido tiene el mismo ID que el Authorize enviado.**

  Pero el Status NO es "*Accepted*".

  Primero se resetea el ***SetInterval()*** y el ***SetTimeout()*** que se pudieron haber ejecutado anteriormente. Luego se notifica al Arduino que la tarjeta fue rechazada, enviando un 4 en la función ***port.write()***.

  Como esta tarjeta fue rechazada, se resetea la variable ***arduino_data***, para poder recibir un nuevo código de tarjeta. Y se pasa al estado E2, para que en el siguiente intervalo de tiempo "*time_main*" se espere una nueva tarjeta, para enviar un nuevo **Authorize**.

- **Si el mensaje recibido tiene el mismo ID que el Authorize enviado.**

  Y el Status es "*Accepted*".

  Primero se resetea el ***SetInterval()*** y el ***SetTimeout()*** que se pudieron haber ejecutado anteriormente. Luego se pasa al estado E4 para enviar un **StartTransaction**, en el siguiente intervalo de tiempo "*time_main*".

4.1.6. Estado E4: Envío del StartTransaction Request.

En este estado, si no hay error de conexión con el VE (detectado por el Arduino), se espera poder enviar un mensaje **StartTransaction** al servidor de UTE. Por eso, durante el estado E4, se verifica si ***arduino_data*** es cero o no.

En caso de que sea cero, entonces se considera un error temporal, por lo que se resetea la variable ***arduino_data*** (para luego poder leer otro dato) y pasar al estado E2. En el estado E2 se espera a que el Arduino se recomponga - cuando envía un 1- o que sea un error definitivo - el Arduino envía nuevamente un 0, unos 30 segundos después -.

Si no hay código error enviado por el Arduino, entonces se envía un **StartTransaction**. Para lograrlo, primero se modifica el archivo JSON de **StartTransaction** para agregarle el dato de ***tarjeta***. Luego se utiliza una función auxiliar ***medidor.energia(function(resultado){})***, para obtener en la variable **resultado** el dato de energía que mide el medidor en el instante que es solicitado por la Raspberry. (ver sección 4.1.13.2)

Dentro de la anterior función, se utiliza la otra función: ***envio.mensaje(mens,socket,dato)***. Con **mens** = "*ComenzarTransaccion*", para que pueda enviar el mensaje correcto; y el argumento **socket** es el mismo que se utiliza en el programa principal y **dato** = **resultado[0]*1000**.

Finalmente, se asigna a la variable (***main_status***) el valor E5, para que en el siguiente intervalo de tiempo "*time_main*" se ejecute el código del estado E5, para leer la respuesta del servidor al **StartTransaction** enviado.

4.1.7. Estado E5: Lectura StartTransaction Response y Seteo del Envío Periódico del MeterValue.

El servidor deberá responder, enviando un JSON con el mismo ID que tenía el último JSON enviado por la Raspberry. Y en el caso del **StartTransaction**, el servidor deberá responder si el Status es "*Accepted*" o no.

- **Si el mensaje recibido NO tiene el mismo ID que el StartTransaction enviado.**

  Luego de un intervalo de tiempo "*time_response_startTransaction*", se vuelve al estado anterior E4, para enviar un nuevo **StartTransaction** al servidor.

  Para lograr dicho objetivo, se setea por única vez la función ***SetInterval()***, que va a ejecutar las líneas de código que contenga dicha función en su interior, cada vez que pase un periodo de tiempo "*time_response_startTransaction*".

  Al mismo tiempo, se setea por única vez, la función ***SetTimeout()***, para contar el pasaje del tiempo. Cuando se llegue al tiempo máximo, seteado en la variable "*time_reset_startTransaction*", entonces se ejecutará una sola vez, el código que contenga dicha función en su interior.

  En este caso, luego de un intervalo de tiempo "*time_reset_startTransaction*", se vuelve al estado inicial E0, para enviar un **BootNotification** al servidor y notificar al Arduino que la Raspberry no está lista. Y también se resetea el ***SetInterval()*** anterior, porque ya no va a ser necesario re-enviar el mensaje **StartTransaction**.

- **Si el mensaje recibido tiene el mismo ID que el StartTransaction enviado.**

  Pero el Status NO es "*Accepted*".

  Primero se resetea el ***SetInterval()*** y el ***SetTimeout()*** que se pudieron haber ejecutado anteriormente. Luego se notifica al Arduino que la transactcion fue rechazada, enviando un 5 en la función ***port.write()***.

  Como esta tarjeta fue rechazada, se resetea la variable ***arduino_data***, para poder recibir un nuevo código de tarjeta. Y se pasa al estado E2, para que en el siguiente intervalo de tiempo "*time_main*" se espere una nueva tarjeta, para enviar un nuevo **Authorize**.

- **Si el mensaje recibido tiene el mismo ID que el StartTransaction enviado.**

  Y el Status es "*Accepted*".

  Primero se resetea el ***SetInterval()*** y el ***SetTimeout()*** que se pudieron haber ejecutado anteriormente.

  Luego se le notifica al Arduino - enviando un 2 en la función ***port.write()***- que el inicio de la transacción fue aceptada y que el Arduino puede habilitar la carga, cuando el vehículo esté en condiciones.

  Además, la respuesta del servidor en el mensaje **StartTransaction** contiene un valor que se utiliza como identificador único de la sesión de carga (**transactionId**). El protocolo OCPP 1.6, requiere que el mismo valor **transactionId** que se recibió del servidor, luego sea enviado dentro del mensaje **StopTransaction** que solicita al servidor detener dicha sesión de carga.

  Mediante la función ***SetInterval()*** se setea el envío periódico de un **MeterValue**, en cada intervalo de tiempo igual al valor de "*time_meterValue*".

  Y lo que se ejecutará, luego de transcurrido dicho intervalo, es el llamado de la función auxiliar ***medidor.energia(function(resultado){})***, para obtener en la variable **resultado** el dato de energía que mide el medidor en el instante que es solicitado por la Raspberry.

  Dentro de la anterior función, se utiliza la otra función:

  envio.mensaje(mens,socket,dato).

  Con **mens** = "*MandarValor*", para que pueda enviar el mensaje correcto; el argumento **socket** es el mismo que se utiliza en el programa principal y **dato** = **resultado[0]*1000**.

  Y la respuesta al **MeterValue** enviado, no será leída en ningún estado en particular, porque la respuesta no generará ningún cambio en el comportamiento del SAVE.

  Finalmente se pasa al estado E6 para, en el siguiente intervalo de tiempo "*time_main*", esperar una nueva tarjeta y validarla, para saber si enviar (o no) un **StopTransaction**.

4.1.8. Estado E6: Envío del StopTransaction Request.

En este estado, se espera recibir algún dato enviado por el Arduino, que puede ser: un código de una tarjeta leída, un código de aviso de que el vehículo se desconectó del SAVE o un código error.

Para la lectura de datos enviados por el Arduino, en la Raspberry se utiliza la función ***parser.on('data', function(data){arduino_data=data})*** que guarda en la variable ***arduino_data*** el dato leído.

Entonces, se verifica la variable ***arduino_data***:

- Si ***arduino_data*** es cero, entonces hay un error en el Arduino, por lo que hay que pasar al estado E8 para enviar inmediatamente un **StopTransaction** al servidor de UTE, sin esperar una lectura de una tarjeta.

- Si ***arduino_data*** es igual a la variable ***tarjeta*** o es un código 3 (el vehículo se desconectó), el primer paso es detener el envío periódico del **MeterValue**, por eso se resetea el correspondiente *SetItnerval()*.

  Luego se envía el mensaje **StopTransaction** al servidor de UTE, utilizando la función auxiliar ***medidor.energia(function(resultado){})***, para obtener en la variable **resultado** el dato de energía que mide el medidor en el instante que es solicitado por la Raspberry.

  Dentro de la anterior función, se utiliza la otra función:

  envio.mensaje(mens,socket,dato).

  Con **mens** = "*DetenerTransaccion*", para que pueda enviar el mensaje correcto; con el mismo **socket** que se utiliza en el programa principal y **dato** = **resultado[0]*1000**.

  Finalmente, se resetea la variable ***arduino_data*** para poder leer un nuevo código y se asigna a la variable (***main_status***) el valor E7, para que en el siguiente intervalo de tiempo *time_main*, se lea la respuesta del servidor al **StopTransaction** enviado.

- Si ***arduino_data*** es un código, distinto a los anteriores, entonces es una tarjeta diferente a la que inició la transacción.

  Como esta tarjeta fue rechazada, se notifica al Arduino - enviando un 6 en la función ***port.write()***- que la tarjeta no es igual a la que inició la transacción.

  Luego se resetea la variable ***arduino_data***, para poder recibir un nuevo código de tarjeta.

  Y no se cambia de estado, porque la idea es mantener la misma transacción abierta y permitir que el vehículo se pueda seguir cargando, hasta que:

  - se pase una tarjeta idéntica a la que inició la transacción.

  - o que el vehículo se desconecte.

  - o el Arduino envíe un código error.

4.1.9. Estado E7: Lectura StopTransaction Response y Detener Envío del MeterValue y del Heartbeat.

El servidor deberá responder, enviando un JSON con el mismo ID que tenía el último JSON enviado por la Raspberry. Y en el caso del **StopTransaction**, el servidor deberá responder si el Status es "*Accepted*" o no.

- **Si el mensaje recibido NO tiene el mismo ID que el StopTransaction enviado.**

  Luego de un intervalo de tiempo "*time_response_stopTransaction*", se va al estado E8, para enviar un nuevo **StopTransaction** al servidor, pero sin esperar la lectura de una tarjeta.

  Para lograr dicho objetivo, se setea por única vez la función ***SetInterval()***, que va a ejecutar las líneas de código que contenga dicha función en su interior, cada vez que pase un periodo de tiempo "*time_response_stopTransaction*".

  Al mismo tiempo, se setea por única vez, la función ***SetTimeout()***, para contar el pasaje del tiempo. Cuando se llegue al tiempo máximo, seteado en la variable "*time_reset_stopTransaction*", entonces se ejecutará una sola vez, el código que contenga dicha función en su interior.

  En este caso, luego de un intervalo de tiempo "*time_reset_stopTransaction*", el estado se mantiene el mismo, pero es necesario notificar al Arduino -enviando un cero en la función ***port.write()***- que la Raspberry no está lista para iniciar una nueva transacción.

- **Si el mensaje recibido tiene el mismo ID que el StopTransaction enviado.**

  Pero el Status NO es "*Accepted*".

  Se resetea el ***SetInterval()*** y el ***SetTimeout()*** que se pudieron haber ejecutado anteriormente. Luego se pasa al estado E6, para que en el siguiente intervalo de tiempo "*time_main*" se espere la nueva lectura de una tarjeta y si es la misma que inicio la transacción, se envía un nuevo **StopTransaction**.

- **Si el mensaje recibido tiene el mismo ID que el StopTransaction enviado.**

  Y el Status es "*Accepted*".

  Primero se resetea el ***SetInterval()*** y el ***SetTimeout()*** que se pudieron haber ejecutado anteriormente. Luego se notifica al Arduino que la transaccion fue detenida, enviando un 3 en la función ***port.write()***.

  Como la transacción fue detenida con éxito, se resetea el campo **transactionId** en el JSON de **StopTransaction**, para que en el estado E1 esto se pueda verificar. Y se resetea la variable ***arduino_data***, para poder recibir un nuevo código de tarjeta.

  Finalmente se pasa al estado E2, para que en el siguiente intervalo de tiempo "*time_main*" se espere una nueva tarjeta, para enviar un nuevo **Authorize**. Y así, poder comenzar una nueva transacción.

4.1.10. Estado E8: Envio StopTransaction Request, Sin Esperar la Lectura de una Tarjeta.

El objetivo, en este estado, es enviar un **StopTransaction** pero sin esperar la lectura de una tarjeta. Así que, el primer paso es detener el envío periódico del **MeterValue**, por eso se resetea el correspondiente *SetItnerval()*.

Luego se envía el mensaje **StopTransaction** al servidor de UTE, utilizando la función auxiliar ***medidor.energia(function(resultado){})***, para obtener en la variable **resultado** el dato de energía que mide el medidor en el instante que es solicitado por la Raspberry.

Dentro de la anterior función, se utiliza la otra función: ***envio.mensaje(mens,socket,dato)***.

Al argumento **mens** = "*DetenerTransaccion*", para que pueda enviar el mensaje correcto; con el mismo **socket** que se utiliza en el programa principal y **dato** = **resultado[0]*1000**.

Finalmente, se resetea la variable ***arduino_data*** para poder leer un nuevo código y se asigna a la variable (***main_status***) el valor E7, para que en el siguiente intervalo de tiempo *time_main*, se lea la respuesta del servidor al **StopTransaction** enviado.

4.1.11. Estado E9: Error Definitivo en Arduino. Se Espera a la Recuperación del Arduino.

A este estado se llega desde el estado E2, solo en el caso que el error detectado por el Arduino se haya mantenido por más de 30 segundos.

En dicho caso, debe detener el envío periódico del **Heartbeat** y del **MeterValue**, por eso se resetean los correspondientes *SetItnerval()*.

Sólo se volverá al estado E0, si se recibe un código de recuperación (un 1) enviado por el Arduino. Si eso sucede, en el estado E0 se enviará el **BootNotification** y así se notificará al servidor que el SAVE vuelve a estar disponible.

4.1.12. Estado Default:

Si la función principal se ejecuta correctamente, pero la variable (***main_status***) no es ningún estado anterior, se imprime un mensaje de error en la consola de la Raspberry.

4.1.13. Funciones Llamadas Durante el Programa Principal.

4.1.13.1. envio.mensaje(mens, socket, dato);

Esta función es utilizada para enviar cada mensaje JSON deseado al servidor de UTE, a partir de un formato base que tiene cada mensaje del protocolo OCPP 1.6.

El procedimiento es: primero seleccionar el formato base y luego, como se sabe la estructura inicial, se puede recorrer cada campo del objeto y modificar el valor de dicho campo.

Finalmente, después de modificar los valores deseados, se envía el JSON, pero en formato string, con la función ***socket.send(JSON.stringify())***

Para seleccionar el mensaje a enviar, hay que asignarle un string a al argumento (**mens**). Este string será el utilizado para distinguir cuál es el JSON que se desea modificar para luego enviar.

El argumento (**socket**), es utilizado en la función que envía el JSON modificado al servidor de UTE. A este argumento simplemente hay que asignarle el mismo websocket utilizado en la función principal ***main()***. Y por último, (**dato**) es el argumento opcional, que se utiliza para asignarle cierto dato específico, al mensaje que se desea enviar.

4.1.13.2. medidor.energia(function(resultado) { });

El archivo **medidor.js** tiene dos funciones definidas: **energia** y **frecuencia_y_FP**. Como en el programa principal, sólo se utiliza la primera, entonces sólo **energia** será descripta.

Dicho archivo importa una librería para NodeJS: "*modbus-serial*" y se declara *client* que será utilizada para habilitar un nuevo puerto de comunicación ModbusRTU.

Luego con la función ***client.connectRTU("/dev/ttyUSB0",{baudRate:9600})*** se especifica el puerto y la velocidad en baudios de la transmisión de datos entre la Raspberry y el medidor de energía.

La función ***energia(callback){}***, devuelve un resultado en la variable **callback**.

La primera linea en dicha función, *client.setID(1)*, define a cuál dispositivo se va a dirigir para pedirle un dato. En este caso, el medidor de energía deberá tener seteado que es el dispositivo 1, porque la Raspberry le solicitará los datos sólo al dispositivo 1.

Luego ***client.readHoldingRegisters(,)*** es para determinar a partir de qué registro empezar a leer y cuántos registros leer de ese en adelante. Lo leído es guardado como un buffer en *energy* y luego en un arreglo **E**[ ] se va guardando flotantes de 32 bits, leyendo dicho buffer como Big Endian.

El arreglo **E**[ ] es el que se devuelve en la instancia **callback** y cuando se llama a la función: **medidor.energia(function(resultado){});** es a la variable **resultado** que se le asigna el valor de **callback**.

4.1.13.3. openSerialPort();

Esta función comienza definiendo **Readline**, que es una propiedad de la librería para NodeJS "*serialport*" que es importada en el archivo principal **SAVE.js** y asociada a la variable **Serialport**.

Luego con la función ***new Serialport('/dev/ttyACM0',{baudRate: 115200})*** se especifica el puerto y la velocidad en baudios de la transmisión de datos entre la Raspberry y el Arduino. Esto queda asociado a **port**.

Mientras que **parser** queda asociado a lo que se recibe en **port** pero delimitado por cada caracter de fin de línea.

A partir de la declaración de la función **openSerialPort()** quedan determinadas:

- **port.write();** En la que la Raspberry envía un dato al Arduino.
- **parser.on('data', function (data) { arduino_data = data });** En la que la Raspberry recibe un dato enviado por el Arduino y lo guarda en la variable (***arduino_data***). El dato puede ser recibido en cualquier momento, por lo que **parser.on()** es un evento que se ejecuta cuando llega un dato enviado por el Arduino, mientras tanto, queda "escuchando" el puerto.

4.1.13.4. connect();

Primero se abre un nuevo **socket** al ejecutar: ***new WebSocket(url,'ocpp1.6')***;
donde se está utilizando la librería para NodeJS "**ws**" que está asociada a *WebSocket*.
La **url** es única para este SAVE, fue determinada por funcionarios de UTE, y está definida como una constante al principio del archivo principal **SAVE.js**.

Como dentro de la función principal ***main()*** se envía al servidor un **Ping**, mediante el comando ***socket.ping()***, se esperaría que si la conexión está estable, el servidor responderá con un **Pong**.

Por eso, cuando suceda dicho evento (la respuesta **Pong**), se va a ejecutar esta linea:
socket.on('pong', function() {this.isAlive=true;}) que valida si el servidor está disponible y en ese caso, asigna a la variable **socket.isAlive** el valor *true*.

Y como dentro de la función principal ***main()***, se setea a la variable **socket.isAlive** el valor *false*, justo antes de enviar al servidor un **Ping**, entonces se sabrá que cuando el servidor no responda, el resultado será que la variable **socket.isAlive** seguirá con el valor *false*.

En el caso que **socket.isAlive** siga con el valor *false*, en la siguiente ejecución de la función ***main()***, se va a ejecutar la siguiente línea de código:

if (socket.isAlive==false){ return socket.close(1000); }

Que como resultado, **cerrará el socket**.

Dentro de la función **connect()** también está la función: ***socket.on('open',function(){});*** que se ejecuta solo cuando se abre el **socket**. En ese caso, se llamará a la función principal ***main()*** a la que se le pasa como argumento el **socket**. Y como ya fue mencionado, dentro de dicha función ***main()***, se valida si el **socket** se mantiene abierto o no.

También, mientras esté abierto el socket, se va a detener el llamado periódico a la función ***socketClose()*** y se resetea el contador que dicha función incrementa.

Mientras el **socket** esté abierto, la Raspberry puede recibir mensajes del servidor en cualquier momento. Por lo que, se utiliza un comando que sólo se ejecutará en caso de que dicho evento ocurra: ***socket.on('message', function incoming(stringJson){}***

Cuando llega dicho mensaje, es guardado en la variable **menssage_recived**, aunque sea un JSON vacío. El mensaje original, llega en formato **string**, y como se desea guardar en un formato JSON, es necesario utilizar la función *JSON.parse()* para cambiar de uno al otro.

El beneficio de tener guardado en formato JSON, es que luego se puede leer cada campo del mensaje por separado, como un objeto.

Se ejecutará ***socket.on('error', function error(error){}*** sólo cuando el **socket** entre en un estado de error; y si eso acontese, se imprime en la consola de la Raspberry un mensaje de error de conexión.

Finalmente, ***socket.on('close',function(){})*** sólo se ejecuta cuando el **socket** entra en estado **close**. Si dicho evento ocurre -por ejemplo, cuando el servidor no responde al **Ping** enviado por la Raspberry- entonces se ejecutan las líneas que están dentro de dicha función:

- Se deja de ejecutar periódicamente la función principal ***main()***, al resetear el (***main_Interval***).

- Luego se llama a la función auxiliar ***socketClose()*** periódicamente. Para lograrlo, se setea la función *SetInterval()*, que va a llamar a dicha funcion auxiliar, cada vez que pase un periodo de tiempo "*time_close*" .

4.1.13.5. socketClose();

Esta función auxiliar es llamada periódicamente en cada intervalo de tiempo "*time_close*", si en algún momento el **socket** paso al estado **close**.

Cada vez que es llamada, incrementa un contador (**cont**) e imprime el valor de la variable en la consola de la Raspberry.

Luego se verifica si esta variable (**cont**) es igual a 10:

- Si es igual a 10, entonces se dejará de enviar **HeartBeat** y **MeterValue**. Se enviará un 0 al Arduino, para avisar que la Raspberry esta fuera de servicio. Y se setea la variable (**main_status**) en EO para que cuando se restablezca la conexión del Websocket, entonces la función principal ***main()*** comience desde el primer estado.

- Si es distinto de 10, simplemente no se ejecutan ninguna de las lineas anteriores.

Y finalmente, vuelve a llamar a la función ***Connect()*** .

4.2. Raspberry Pi y Periféricos

En la presente sección se expondrán las razones por las que se optó por utilizar una Raspberry Pi para implementar la comunicación ente el SAVE y el centro de control de carga, cómo se logró la misma y además se verá como se implemento la comunicación entre la Raspberry, el analizador de red y el Arduino.

4.2.1. Preparación del Entorno de Desarrollo en la Raspberry Pi.

Para la implantación de la comunicación entre el SAVE y el servidor del centro de control de carga (servidor de UTE) se optó por utilizar una Raspberry Pi 3 B+ y emplear lenguaje NodeJS (basado en JavaScript) mediante la ejecución de un cliente Websocket.

En esta sección se explican en detalle cómo preparar, desde cero, el entorno de desarrollo necesario para implementar la comunicación con el servidor.

En primera instancia se debe tener en cuenta que una Raspberry es una computadora con muy buenas prestaciones si se tiene en cuenta su reducido tamaño y costo, como tal la misma funciona en base a un sistema operativo que se obtiene de forma gratuita en la web oficial de Raspberry[4].

Para instalar el sistema operativo es necesario contar con una tarjeta SD de al menos 6 GB. Como se mencionó anteriormente, el sistema operativo se instala siguiendo las instrucciones dadas en la web oficial de Raspberry. Hay distintas opciones de sistema operativo a instalar, y de cómo hacerlo, en este caso se descargó el asistente de instalación para Windows, Raspberry pi imagen v1.2 y se instaló la versión de Debian recomendada.

Una vez instalado el sistema operativo y verificado su funcionamiento se procedió a la instalación de NodeJS y el gestor de paquetes npm. A continuación se exponen algunas de las razones por las que se optó por esta solución.

En primera instancia la elección de NodeJS como entorno de desarrollo se debió a la posibilidad de obtener una comunicación con el servidor del centro de control de carga de fácil implementación, como ya se explicó anteriormente la misma es a través Websocket. NodeJS permite hacerlo de forma muy compacta y sencilla a través de la librería WS.

Otra de las razones por las que se optó por NodeJS es que se trata de un software libre, por lo que, es de fácil acceso y se dispone de mucha información sobre su utilización. También es importante notar que mediante el lenguaje JavaScript es posible implementar la comunicación con los componentes periféricos como ser el medidor de energía y el Arduino de forma sencilla.

No menos importante es el hecho de poder inicializar de forma sencilla la ejecución del servidor al iniciar o reiniciar la Raspberry, para lograrlo se utilizó el gestor de procesos pm2. Para obtener información de su utilización se tiene la web oficial[5], donde se encontrará en detalle como instalarlo y realizar las configuraciones necesarias para lograr que se ejecute el proceso al encender el SAVE.

Para obtener los detalles de cómo instalar Nodejs y el gestor de paquetes npm puede remitirse al anexo A.4. También se encontrará información de cómo comenzar a utilizar el gestor de paquetes npm y como proceder para la creación de un nuevo proyecto utilizando estas herramientas.

[4] http://raspbian.org/

[5] https://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/

4.2.2. Comunicación Entre la Raspberry Pi y el Arduino.

La comunicación entre la Raspberry y el Arduino se implementó a través del puerto USB.

Como se menciona anteriormente se utilizo la distribución de linux Debida, por lo que la dirección del puerto viene dada por /dev/ttyACM0 o /dev/ttyUSB0 dependiendo del protocolo de comunicación que utilice el periférico.

Dado que se utilizó un Arduino que requiere un direccionamiento del tipo ACM fue necesario modificar los permisos de acceso al puerto USB. Por defecto Debian reconoce la dirección de un dispositivo cuando la misma es del tipo /dev/ttyUSB0, si se trata de un dispositivo que utiliza un direccionamiento del tipo /dev/ttyACM0 es necesario modificar los permisos de acceso al puerto USB.

Para eso es necesario ejecutar en la terminal el siguiente comandado *sudo chmod 777 /dev/ttyACM0*, seguido de *sudo usermod -a -G dialout*, donde "usermod" es el nombre del dispositivo que está ejecutando el sistema operativo. De esta forma con la primer instrucción se habilitan los permisos y con la segunda se configuran como permanente los cambios.

Una ves hecho lo expuesto anteriormente mediante el código presentado a continuación (explicado en la sección 4.1.13.3) es posible abrir la comunicación entre la Raspberry y el Arduino:

```
function openSerialPort(){

 const Readline = Serialport.parsers.Readline;
    port = new Serialport('/dev/ttyACM0', {baudRate: 115200});
    parser = port.pipe(new Readline({ delimeter: '\r\n'}));
}
```

De esta forma queda de manifiesto el procedimiento seguido para configurar la Raspberry para que se comunique con el Arduino. Desde el Arduino no será necesario más que una linea de código, para hacer posible dicha comunicación.

4.2.3. Conexión de la Raspberry a una Red VPN Mediante WiFi.

La conexión con el servidor del centro de control de carga se realizó mediante una red VPN -una red privada de acceso restringido- la misma se realizó mediante un tarjeta SIM proporcionada por UTE para tal propósito. Dado que la Raspberry no tiene lector de tarjeta SIM fue necesario utilizar un dispositivo adicional para lograr la conexión a la red.

En principio se realizaron las pruebas colocando la tarjeta SIM en un celular, logrando así la conexión a la red VPN. Una vez lograda la conexión a la red desde el celular, este comparte la conexión con la Raspberry mediante WiFi y se tiene así acceso a la red desde la Raspberry.

El propósito de las pruebas descriptas fue que, de ser posible la conexión de la Raspberry a la red de la forma descripta, se podría remplazar luego el celular por un módem que permita la lectura de tarjeta SIM y genere una red WiFi o en su defecto contara con conexión cableada RJ45.

Tras diversas pruebas con distintos tipos de módems se opto por el HUAWEI E8372 ya que permite una comunicación WiFi con la Raspberry y puede ser alimentado desde una fuente

independiente mediante una conexión USB, lo que permite una mayor estabilidad en la conexión que si se conecta directamente del puerto USB del a Raspberry. Se estima que el consumo de corriente es determinante para lograr estabilidad en la conexión por lo que tomar la alimentación directamente del puerto USB no es suficiente. Para lograr que el dispositivo con la tarjeta SIM se conecte a la red VPN fue necesario acceder al administrador de redes mediante la dirección IP 192.168.8.1, donde usuario y contraseña deben ser admin. Una vez ingresado al administrador de redes debe realizarse la configuración con los datos mostrados en la imagen de la figura 4.2a, donde contraseña debe ser "root". Una vez configurado el perfil de red presionando siguiente, se pasa a las configuraciones WLAN como se puede ver en la figura 4.2b, donde en el campo clave compartida inicial WAP se utilizó "v2g12345678".

(a) Configuración de Perfil.

(b) Configuraciones WLAN.

Figura 4.2: Configuración de la conexión a la red APN.

Luego de ingresados todos los campos se debe presionar finalizar para aplicar la nueva configuración a la conexión de red. De esta forma se puede acceder a la red desde cualquier dispositivo que así lo permita mediante comunicación WiFi con el módem HUAWEI utilizado.

4.2.4. Comunicación Entre la Raspberry y el Medidor de Energía.

Para obtener la medida de la energía suministrada por el SAVE al vehículo durante la carga se optó por utilizar un analizador de red con comunicación Modbus-RTU sobre un canal RS-485. El mismo permite obtener todas las magnitudes eléctricas de interés como ser tensión, corriente, energía, potencia activa y reactiva.

La comunicación entre la Raspberry y el analizador de red, para obtener los valores de las magnitudes de interés, se realizó mediante un adaptador USB-RS485 modelo CH340 para el cual fue necesario instalar el controlador adecuando para Debian (ver Anexo A.5).

Una vez instado el controlador desde el servidor NodeJS es posible obtener la medida de las magnitudes de interés sin mayor dificultad mediante la utilización de la librería Serialport instalada mediante el gestor de paquetes npm.

A continuación se presenta el código JavaScript implementada para obtener la medida de energía, que fue explicado anterior en la sección 4.1.13.2.

```
var ModbusRTU = require("modbus-serial");
var client = new ModbusRTU();

client.connectRTU("/dev/ttyUSB0", { baudRate:9600});

function energia(callback) {

console.log("Entre a mandar la energia");
client.setID(1);  . // read the 8 registers starting at address 71 on device number 1

    client.readHoldingRegisters(71,8).then( (energy) => {
        const buf = energy.buffer;
        var E= [];
        var i= 0 ;
        var j= 0 ;
        while (i<=12){
       E[j] = buf.readFloatBE(i)/1000;
           i= i+4;
           j= j+1;
        }
callback(E);
});
}
```

Con lo expuesto en esta sección fue posible obtener la medida de la energía que debe ser enviada al centro de control de carga al iniciar o finalizar una sección de carga, así como cada cierto tiempo durante la misma.

CAPÍTULO 5

PROTOCOLO IEC: COMUNICACIÓN SAVE-VEHÍCULO.

5.1. Introducción.

En el presente capítulo se expone las principales características de la norma IEC 61851-1 que en el Anexo A (ver [7]), define la comunicación entre el SAVE y el vehículo eléctrico. Esta comunicación es idéntica a la definida en la norma SAE J1772 oct. 2017 (ver [8]).

A continuación se describe el protocolo de comunicación que propone la mencionada norma para un tipo de carga semi-rápida y rápida, ya que son las implementadas en el SAVE. La comunicación entre el SAVE y el VE se realiza a través de la conexión entre el borne **Control Pilot** del conector Tipo 2 y la tierra.

La definición de estados del SAVE y el VE, según el voltaje y la señal PWM (que se analizará con detalle en este capítulo), se presenta en la siguiente tabla 5.1.

Designación de Estado	V_{SAVE} V_{dc} Nominal	V_{VE} V_{dc}Nominal	Descripción de SAVE/VE
Estado A	12,0 V	0 V	Sin vehículo conectado
Estado B1	9,0 V	9,0 V	Vehículo conectado, voltaje constante, el SAVE y el VE no están listos para la transferencia de energía.
Estado B2	9,0 V	9,0 V	Vehículo conectado, pwm de 1 kHz, el SAVE listo, pero el VE no está listo para la transferencia de energía.
Estado C	6,0 V	6,0 V	Vehículo conectado, SAVE y VE listos para la transferencia de energía, no requiere ventilación.
Estado D	3,0 V	3,0 V	Vehículo conectado, SAVE y VE listos para la transferencia de energía, requiere ventilación.
Estado E	0 V	0 V	SAVE fuera de servicio y vehículo desconectado.
Estado F	-12 V	-12 V	Otros problema en el SAVE.

Tabla 5.1: Secuencia de Estados Según el Protocolo IEC o SAE.

El protocolo IEC 61851-1 en el Anexo A (o SAE J1772) también establece que el cargador debe informarle al vehículo cuál es su capacidad de carga máxima, para esto propone la utilización de una señal PWM de $\pm 12\ V$ y frecuencia 1 kHz con un ciclo de trabajo variable. Para ello propone para el CP (Control Pilot) el circuito de la figura 5.1 para la comunicación entre el SAVE y el VE. Cuando en el pin CP hay 12 V y se conecta un VE debido a la resistencia R3 (situada en el VE) y con S2 abierto el valor de la tensión en el CP cae a 9 V. Si el VE cierra la llave S2 debido al paralelo de las resistencias R2 y R3 la tensión en el CP cae a 6 V. Los valores de tensión presentados en la tabla anterior se obtienen gracias a las resistencias R1, R2 y R3.

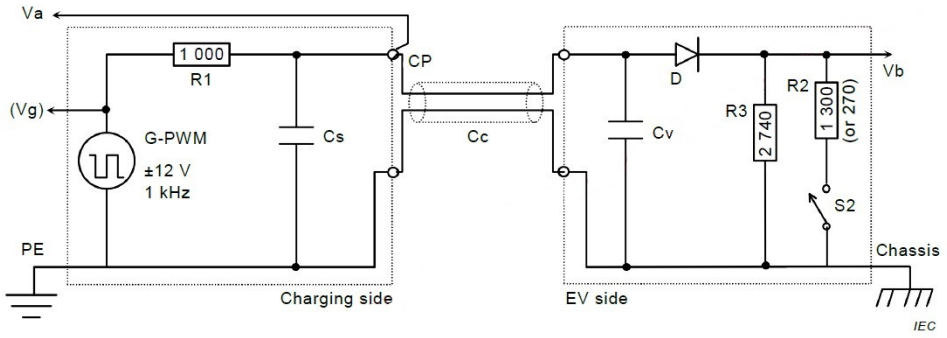


Figura 5.1: Circuito para el CP según IEC.

En la tabla 5.2 se presenta la relación entre el duty cycle y la corriente máxima admitida para la carga.

SAVE Duty Cycle Nominal.	Entrada al Vehículo.	Máxima Corriente a Extraer por el Vehículo.
Duty Cycle = 0 %	Duty Cycle < 3 %	Estado Fo E carga no permitida.
Duty Cycle = 5 %	4,5 % ≤ Duty Cycle ≤ 5,5 %	Indicio que com. digital es necesaria.
	7 % < Duty Cycle < 8 %	Estado de error, Carga no permitida.
	9,5 % ≤ Duty Cycle ≤ 10 %	6 A.
10 % ≤ Duty Cycle ≤ 20 %	10 % ≤ Duty Cycle ≤ 20 %	Corriente Max = (Duty Cycle %)×0,6.
20 % < Duty Cycle ≤ 85 %	20 % < Duty Cycle ≤ 85 %	Corriente Max = (Duty Cycle %) ×0,6.
85 % < Duty Cycle ≤ 96 %	85 % < Duty Cycle ≤ 96 %	Corriente Max = (D.C. % - 64) ×2,5.
	96 % < Duty Cycle ≤ 96,5 %	80 A.
Duty Cycle = 100 %		Estado B1, C1 o D1 carga no permitida.

Tabla 5.2: Duty Cycle del PWM vs Corriente de Carga Máxima.

Cuando un conector es conectado al SAVE, éste debe poder determinar cual es la corriente máxima que el cable soporta. Es decir, si el cable soporta un máximo de 13 A el SAVE no puede permitir una carga a mayor corriente. Para poder determinar la corriente que soporta cada cable el SAVE chequea el valor de la resistencia R_c, la cual esta conectada en el conector entre el pin de proximidad (PP) y tierra como se puede ver en el figura 5.2.

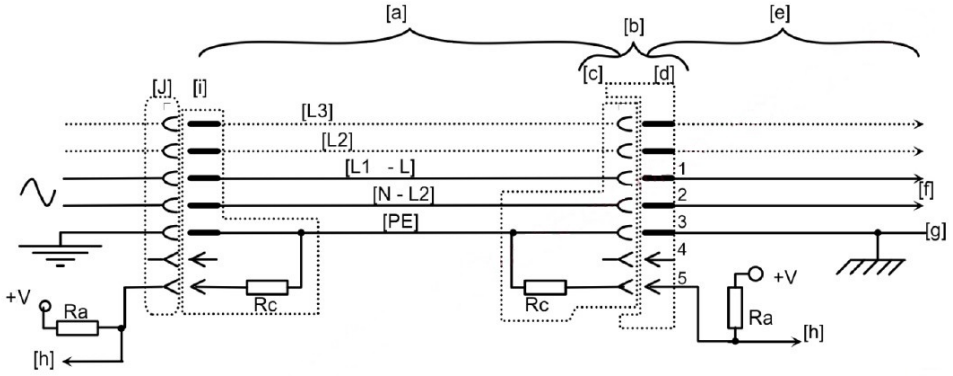


Figura 5.2: Circuito Rc.

Los valores que puede tomar la resistencia R_c están establecidos por la norma IEC 61851-1. La siguiente tabla muestra los posibles valores para la resistencia R_c.

Corriente máxima del cable (A)	Resistencia nominal R_c (Ω)	Mínima disipación (W)	Rango de valores para R_c (Ω)
	Error o sin conector		> 4500
13	1500	0,5	$1100 - 2460$
20	680	0,5	$400 - 936$
32	220	1	$164 - 308$
63(3-fases)/70(1-fase)	100	1	$80 - 140$
	Error		< 60

Tabla 5.3: Valores posibles para la resistencia R_c.

5.2. Descripción del Protocolo de Carga de un VE.

Este protocolo está definido en la norma europea IEC 61851-1, que en Uruguay es la norma UNIT-IEC 61851-1:2010. Esta norma define las condiciones de los estados y de las transiciones que ocurren en un proceso de carga de un vehículo eléctrico. La siguiente imagen muestra el diagrama de flujo de los estadios y transiciones,lo que se detalla a continuación.

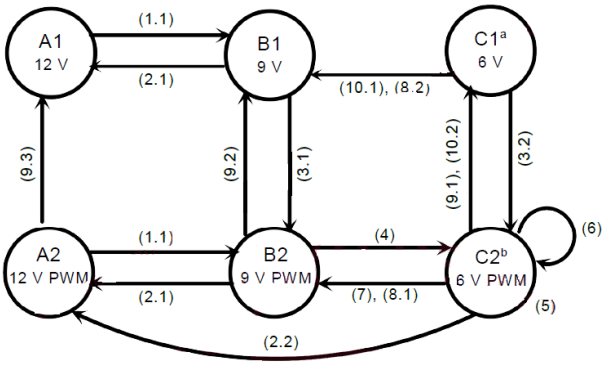


Figura 5.3: Diagrama de la comunicación IEC o SAE.

Estado A1:

Este estado es el punto de partida de la comunicación, donde no hay aún ningún vehículo eléctrico conectado al SAVE. En este estado el SAVE debe garantizar en el pin de control de su conector una señal continua de 12 V. En todo momento el SAVE chequea que el voltaje en el pin de control sea 12 V ya que cuando se conecta un VE éste valor pasa a ser de 9 V (debido a las resistencias internas que coloca el VE). Esta transición se puede ver en la figura 5.3 como la numero 1.1 que da paso al estado B1.

El cambio en la tensión en el pin de control se puede ver en la figura 5.4.

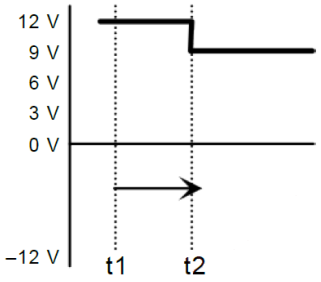


Figura 5.4: Transición 1.1.

Estado B1:

En este estado el VE se encuentra ya conectado provocando que el valor de tensión en el pin de control sea de 9 V. Al reconocer este cambio el SAVE chequea (como se verá con detalle más adelante) la corriente máxima que soporta el cable con el cual se realizó la conexión entre el SAVE y el VE. Una vez que obtiene este valor lo compara con el valor de corriente máxima que el SAVE es capaz de suministrar para tomar la decisión de con qué valor de corriente máxima se

podrá realizar la carga. Por ejemplo, si el SAVE es capaz de suministrar una corriente de 32 A pero el cable que se utilizó para conectar el vehículo soporta 16 A el SAVE debe comunicarle al vehículo (por medio del PWM) que la carga se va a realizar a un máximo de 16 A, para que éste ajuste sus resistencias de modo de consumir a lo sumo dicho valor de corriente (el cálculo del PWM se verá con detalles más adelante). Una vez que el SAVE tiene el valor del PWM se realiza la transición 3.1 que da lugar al estado B2.

La figura 5.5 muestra este cambio en la tensión del pin de control.

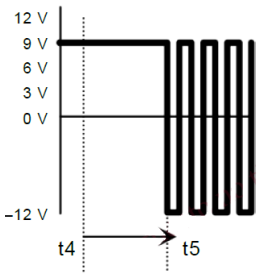


Figura 5.5: Transición 3.1.

La transición 2.1 se llevaría a cabo en el caso que el vehículo eléctrico sea desconectado del SAVE, lo que llevaría al sistema nuevamente al estado A1.

Estado B2:

Una vez que el SAVE está enviando la señal PWM adecuada para la carga el VE debe interpretar esta señal y realizar el ajuste de sus resistencias internas para consumir corriente sin superar la máxima que corresponde al valor del PWM. Cuando el vehículo está pronto para la carga cambia el valor de las resistencias conectadas al pin de control llevando la tensión de 9 V a 6 V, dando lugar a la transición número 4 que conduce al estado C2.

La figura 5.6 muestra el cambio de la tensión en el pin de control.

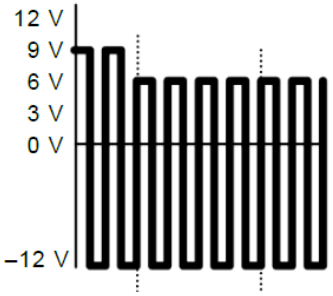


Figura 5.6: Transición 4.

La transición 2.1 se produce cuando se desconecta el vehículo eléctrico.

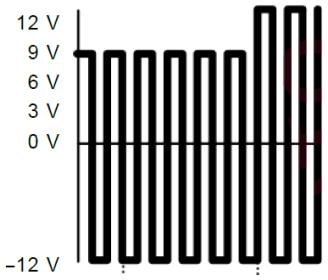


Figura 5.7: Transición 2.1.

La transición 9.2 se da cuando el SAVE deja de enviar la señal PWM al vehículo informándole que no está disponible para cargar.

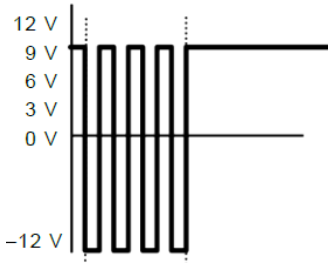


Figura 5.8: Transición 9.2.

Estado C2:

Este estado es donde se realiza la carga del VE, por lo tanto el sistema se mantendrá en este estado mientras la carga transcurra con normalidad. Si el vehículo decide cortar la carga se realiza la transición 7 que vuelve al sistema al estado B2.

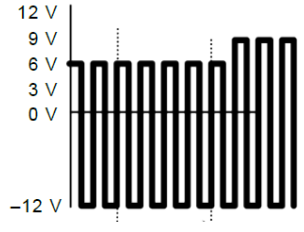


Figura 5.9: Transición 7.

Si la carga se corta por decisión del SAVE se realiza la transición 9.1 llevando al sistema al estado C1.

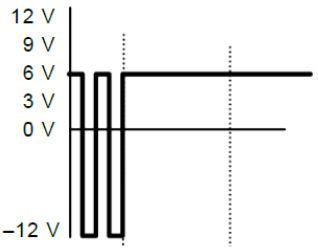


Figura 5.10: Transición 9.1.

Si se desconecta el vehículo del SAVE se realiza la transición 2.2 hacia el estado A2 cortándose la carga inmediatamente.

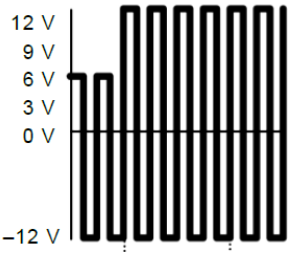


Figura 5.11: Transición 2.2.

Estado C1:

A este estado se llega cuando el SAVE debe cortar la carga del vehículo, ya sea porque el usuario detuvo la carga, porque se le envió la orden del centro de control de carga que cortara la carga, etc. Este es un estado de transición ya que cuando el vehículo reconoce que ya no se envía más el PWM se realiza la transición 8.2 donde el vehículo cambia sus resistencias para lograr en el pin de control un valor de 9 V llevando al sistema al estado B1.

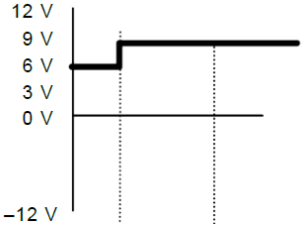


Figura 5.12: Transición 8.2.

Si el SAVE vuelve a generar el valor de PWM antes que ocurra la transición 8.2 se vuelve al estado de carga C2 con la transición 3.2.

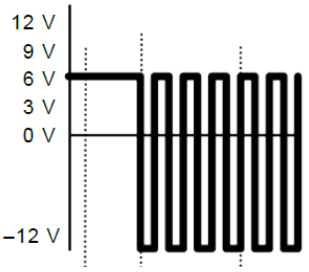


Figura 5.13: Transición 3.2.

Estado A2:

A este estado se llega cuando el vehículo se desconecta del SAVE, por lo tanto, en este caso el sistema se debe asegurar cortar el suministro de energía y dejar de generar la señal PWM, en la transición 9.3, dejando al sistema en el estado A1.

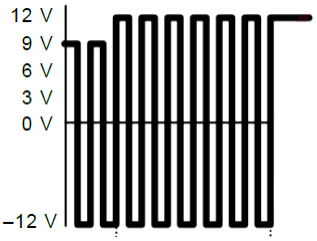


Figura 5.14: Transición 9.3.

CAPÍTULO 6

CIRCUITO IMPLEMENTADO SEGÚN PROTOCOLO IEC.

En el presente capitulo se presenta el circuito diseñado para cumplir con los requerimientos del protocolo IEC 61851-1 en Anexo A (o SAE J1772) descrito en el capitulo anterior.

Se detalla el procedimiento seguido para lograrlo, los resultados de las simulaciones realizadas para verificar su funcionamiento y los resultados obtenidos luego de su implementación.

6.1. Criterios de Diseño del Circuito IEC.

Se utilizará un Arduino Uno para la generación de una señal PWM de 1 kHz con duty cycle ajustable mediante el mismo Arduino.
Se busca obtener una salida de $\pm 12\ V$ a partir de una señal de 0 V a 5 V.
El tiempo de subida y bajada de la señal no debe superar los 2 us, siendo este desde el 10 % hasta el 90 % del cambio de amplitud de la señal.
La salida debe ser 12 V cuando se tiene 0 V a la entrada.
Se utilizará un puente inversor implementado con un pMos y un nMos.
Se utilizarán dos fuentes de 12 V.

6.2. Descripción del Circuito Implementado.

En la figura 6.1 se presenta el circuito que se utilizó para la comunicación entre el SAVE y el vehículo eléctrico. El mismo funciona de acuerdo a lo requerido por el protocolo IEC, comandado por una señal PWM de 0 V a 5 V con un duty cycle (ciclo de trabajo) generado y ajustable por un Arduino UNO.

En términos generales el comportamiento del circuito se puede resumir en que se obtiene a la salida una señal PWM de $\pm 12\ V$ a partir de una señal de 0 V a 5 V en la entrada. Teniendo la salida el mismo duty cycle que la señal de entrada. Vale aclarar que cuando se hace referencia a la salida de $\pm 12\ V$ se esta considerando que la misma está en vacío.

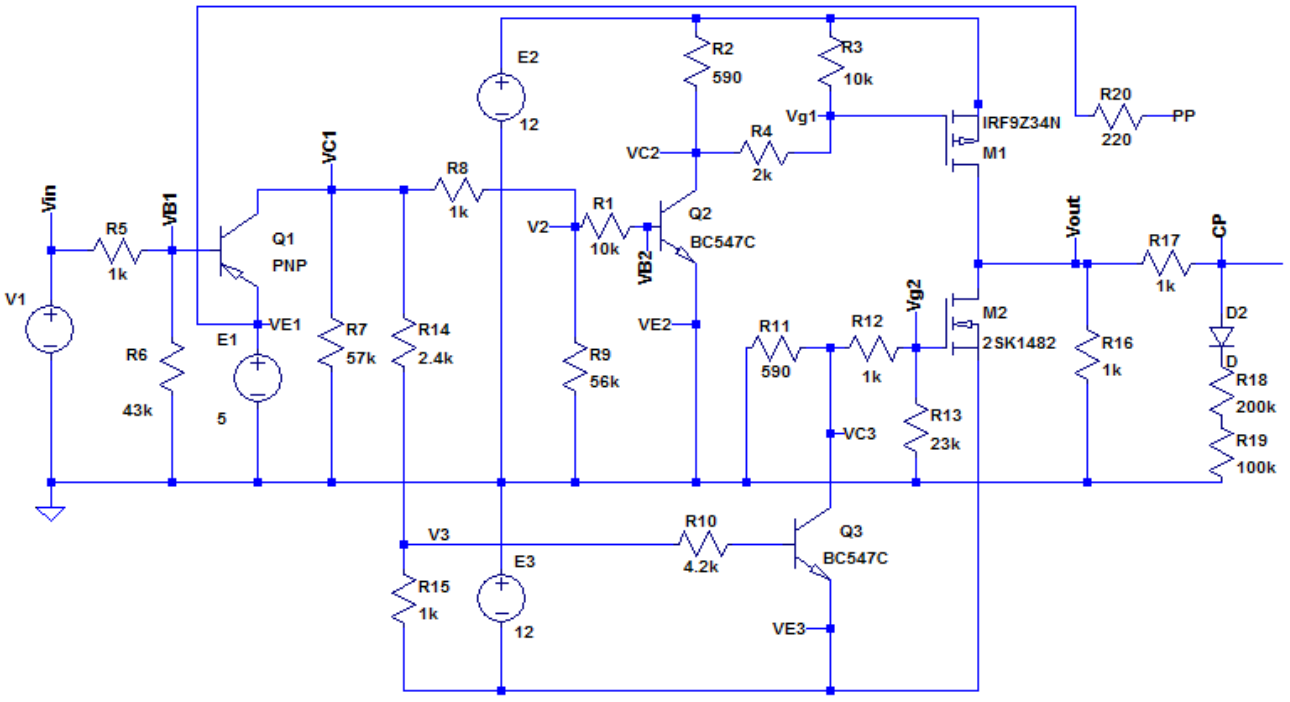


Figura 6.1: Circuito para la comunicación Save-Vehículo según IEC o SAE.

Para obtener la salida de $\pm 12\ V$ se utilizo una rama inversora, implementada con un transistor PMOS (M1) y un NMOS (M2), referida al punto medio de dos fuentes de $12\ V$ conectadas en series como se observa en la figura 6.2.

El circuito de control de los mosfet esta diseñado para que los mismos operen en zona de corte o en zona lineal dependiendo del valor de la entrada. Con el control se logra que la salida sea $12\ V$ cuando la señal de entrada está en $0\ V$ y $-12\ V$ cuando está en $5\ V$.

Este modo de operación se debe a que la salida debe ser $12\ V$ constante cuando no haya vehículo conectado al cargador, de esta manera la señal de control permanece en cero mientras el cargador no está siendo utilizado.

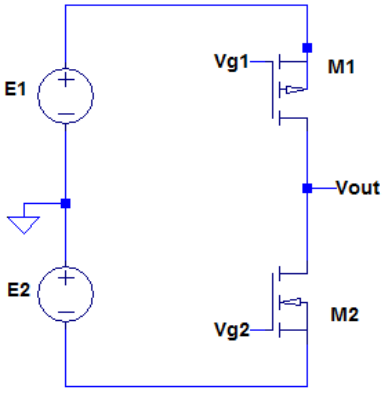


Figura 6.2: Circuito IEC Simplificado.

En este punto no se detallará las características constructivas de un transistor mosfet ni las diferencias de funcionamiento [1], solo se mencionará que es un componente de circuito que tiene

[1] http://hispavila.com/wp-content/uploads/2015/08/tema4$_e l - mosfet.pdf$

tres terminales denominadas gate (G), drain (D) y source (S) a partir de las cuales es posible controlar el funcionamiento del componente según la polarización del mismo.

Se debió diseñar el control para lograr tener a la salida 12 V cuando la señal enviada por el Arduino sea 0 V y tener -12 V cuando la señal de entrada sea 5 V. Para cumplir con el requerimiento descripto fue necesario controlar los mosfet para que pasen de un estado encendido (ON) a un estado apagado (OFF). Lo expuesto anteriormente se puede resumir según la tabla 6.1.

V_{in} (V)	pMos	nMOs	V_{out} (V)
0	ON	OFF	+12
5	OFF	ON	-12

Tabla 6.1: Estado de los Mosfet.

En una instancia previa al diseño del control se determinó el valor de carga máxima que debe permitir el puente inversor y así determinar los requerimientos mínimos para los transistores mosfet. Teniendo en cuenta lo explicado en el capitulo 5 sobre el protocolo IEC y las resistencias que configura el vehículo según su estado en el proceso de carga se determinó que la corriente máxima requerida en la comunicación viene dada por la ecuación 6.1.

$$i_{out} = \frac{V_{out}}{R_{eq_{min}}} = \frac{12}{330} = 36\ mA \tag{6.1}$$

Por lo tanto se establece como requerimiento de corriente un mínimo para los mosfet de 200 mA, por otro lado se tiene que la tensión drain-source (V_{DS}) máxima a la que estarán sometidos los transistores será de 24 V por lo que se establece como requerimiento mínimo de tensión $V_{DS_{min}} = 30\ V$.

Teniendo presente los requerimientos mínimos para los transistores que conformarán el puente se procedió a buscar en el mercado local la disponibilidad de transistores con los que se pueda realizar el diseño del circuito de comunicación IEC. Dada la limitada oferta con que se contó se terminaron definiendo para la realización del puente inversor un transistor pMos IRF9Z34N de 19 A, 55 V (M1) y un nMos SK1482 de 1,5 A, 30 V (M2), ambos con encapsulado TO.

Una vez definidos los transistores se realizó un análisis de las características más relevantes para lograr el funcionamiento deseado. Obteniéndose los valores resumidos en la tabla 6.2.

Transistor	V_{to}	$R_{DS_{on}}$
pMos	$-4{,}1$	0,10 Ω
nMos	1,8	0,15 Ω

Tabla 6.2: Características de los Mosfet.

Luego se pasó a analizar las condiciones que deben verificarse para lograr su funcionamiento según lo visto en la tabla 6.1. En este punto no se entrará en detalle respecto a las características y modos de funcionamiento de un transistor mosfet, solo se mencionará que los mismos pueden

*http : //electronica.ugr.es/ amroldan/deyte/cap*05.*htm*

trabajar en tres zonas distintas de operación, denominadas zona de corte, zona lineal y zona de saturación.

6.3. Diseño del Circuito de Comunicación SAVE-VE.

En primera instancia se pretende lograr el funcionamiento del puente haciendo que ambos mosfet conmuten su estado de operación entre la zona de corte y la zona lineal, para ello se debe asegurar que la polarización de sus terminales cumplan con lo presentado en la tabla 6.3.

Transistor	Zona de Corte	Zona Lineal
NMOS	$V_{gs} \leq V_{T_o}$, $V_{ds} > 0$	$V_{gs} > V_{T_o}$, $0 < V_{ds} < V_{gs} - V_{T_o}$
PMOS	$V_{gs} \geq V_{T_o}$, $V_{ds} < 0$	$V_{gs} < V_{T_o}$, $0 > V_{ds} > V_{gs} - V_{T_o}$

Tabla 6.3: Condiciones de operación Mosfet.

Teniendo en cuenta lo expuesto anteriormente en esta sección es posible comenzar el diseño del circuito de control.

En primera instancia se tiene que la tensión source del transistor M1 será siempre 12 V mientras que la tensión source del transistor M2 será -12 V. Por lo tanto para lograr que ambos transistores conmuten entre las dos zonas de operación deseadas será necesario controlar sus tensiones de gate.

Para el control de la tensión de gate del transistor M1 se diseñó el circuito mostrado en la figura 6.3.

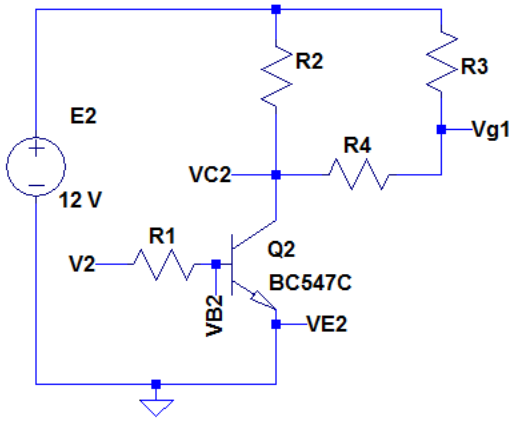


Figura 6.3: Control del pMos.

Para lograr variar la tensión V_{g_1} será necesario que el transistor de unión bipolar npn Q_2 también funcione como llave, conmutando entre los estados ON/OFF. Para que un transistor bipolar se comporte como una llave, éste debe operar entre la zona de corte y la zona de saturación. Por lo que será necesario controlar la corriente de base, lo que se puede lograr haciendo variar la tensión V_2 en el extremo de R_1.

Para tal propósito se optó por un transistor BC547C, a partir de su hoja de datos se define como punto de funcionamiento para la región de saturación una corriente de base de 1 mA y corriente colector-emisor de 20 mA lo que da lugar a un caída de tensión entre colector-emisor

de 0,2 V, además con este valor de corriente de colector, se tiene que la potencia disipada en la resistencia R_2 es de 0,24 W.

Asumiendo una tensión de control $V_2 = 5\ V$ y teniendo en cuanta que en saturación la tensión base-emisor es de 0,8 V se tiene que la diferencia de tensión en R_1 es de 4,2 V, por lo que para lograr una corriente de base de 1 mA, R_1 debe ser igual a 4,2 $k\Omega$.

Para tener una corriente colector-emisor de 20 mA con una tensión colector-emisor de 0,2 V será necesario una resistencia de colector $R_2 = 600\ \Omega$, despreciando la caída de 0.2 V para el cálculo se tiene $R_2 = 590\ \Omega$. Luego haciendo que la tensión de control sea cero o negativa lleva al transistor Q_2 a la zona de corte. En esta condición la tensión base-emisor es cero y la tensión colector emisor es 12 V.

En la tabla 6.4 se presenta una síntesis de los resultados obtenidos en el análisis precedente para el funcionamiento del transistor Q_2.

Estado	$V_2(V)$	$V_{BE}(V)$	$V_{CE}(V)$	$I_B(mA)$	$I_C(mA)$	$R_1(k\Omega)$	$R_2(\Omega)$
Saturación (ON)	5,0	0,8	0,2	1,0	20,0	4,2	590
Corte (OFF)	0	0	12	0	0	− − −	− − −

Tabla 6.4: Estados de operación de Q_2.

En estas condiciones se tendrá una tensión V_{C_2} igual a 0,2 V cuando Q_2 esté en saturación, y será de 12 V cuando esté corte.

Para este análisis se despreció la influencia de las resistencias R_3 y R_4 , ya que eligiendo las mismas de forma tal que su serie sea mucho mayor que R_2, el paralelo de R_2 con la serie de ambas tenderá a R_2.

Para fijar el valor de las resistencias R_3 y R_4 será necesario tener en cuenta las relaciones planteada en la tabla 6.3 para el funcionamiento del pMos y el estado de funcionamiento de Q_1 según la tabla 6.4.

Partiendo de Q_2 en corte se tiene que la tensión V_{g_1} es igual a 12 V por lo tanto la tensión $V_{gs} = 0\ V \geq -4{,}1\ V = V_{t_0}$, por otra parte observando que la tensión de drain en el circuito de la figura 6.2 va a ser cero o negativa ya que $M1$ no está en zona líneas según las condiciones mostradas en la tabla 6.3 se concluye que $V_{ds} < 0V$, por lo tanto en estas condiciones $M1$ está en corte. De los resultados obtenidos en el análisis previo no se desprenden los valores requeridos para R_3 y R_4.

Al pasar Q_2 a saturación se tiene que V_{g_1} toma su valor según el divisor de tensión dado por R_3 y R_4, si se desprecian los 0,2V del punto V_{c_2} se tiene que V_{g_1}viene dado por la ecuación 6.2.

$$V_{g_1} = \frac{R_4}{R_3 + R_4} * E_2 \tag{6.2}$$

Por lo tanto ajustando el valor de las resistencias del divisor de tensión se logrará tener la tensión de gate apropiada para que el transistor M1 pase a trabajar en la zona lineal. Según la tabla 6.3 se debe verificar que la tensión $V_{gs} < V_{t_0} = -4{,}1\ V$ por lo tanto ajustando el divisor resistivo formado por R_3 y R_4 a una relación de un medio se tiene que la tensión $V_{gs} = -6\ V$ y se verifica una de las condiciones necesarias para que M1 este en zona lineal. Sin embargo

al analizar la curva $I_d = f(V_{ds})$ para distintas tensiones gate-source se determinó que el mejor punto de operación para este caso es con $V_{gs} \approx -10\ V$ por lo que se definió $R_3 = 10\ k\Omega$ y se calculo $R_4 = 2\ k\Omega$.

Resta verificar la otra condición necesaria para que M1 opere en la zona lineal según lo expuesto en la tabla 6.3. Para esto si se asume que el transistor M1 está conduciendo y que la condición de carga más desfavorable para la caída de tensión drain-source es cuando $i_{out} = 36\ mA$, se tiene que $0 > V_{ds} = -3{,}6\ mV > V_{gs} - V_{t_0} = -1{,}9\ V$ dado que $R_{DS_{on}} = 0{,}10\ \Omega$. Por lo tanto se verifica la otra condición necesaria para que M1 este en la zona lineal.

En este punto es válida la observación de la importancia de haber utilizado un transistor con una resistencia $R_{DS_{on}}$ lo suficientemente pequeña para que la caída de tensión drain-source pueda ser despreciable frente a los $12\ V$, cumpliéndose que la salida sea la requerida en el semi-ciclo positivo de la señal PWM con un consumo mínimo de potencia.

Antes de pasar al análisis de la rama del circuito que controla al nMos se analizará el circuito desarrollado para invertir la señal PWM de entrada, generada por el Arduino. Este circuito se diseñó e implementó para lograr que el PWM de salida sea $12\ V$ cuando la señal de entrada es $0\ V$ y la salida sea $-12\ V$ cuando la entrada es $5\ V$.

En la figura 6.4 se muestra el circuito implementado. Lo que se buscó cuando se propuso este circuito fue obtener $V_{c_1} = 5\ V$ cuando la señal de entrada es $0\ V$ y tener un $V_{c_1} = 0\ V$ al ser $V_{in} = 5\ V$.

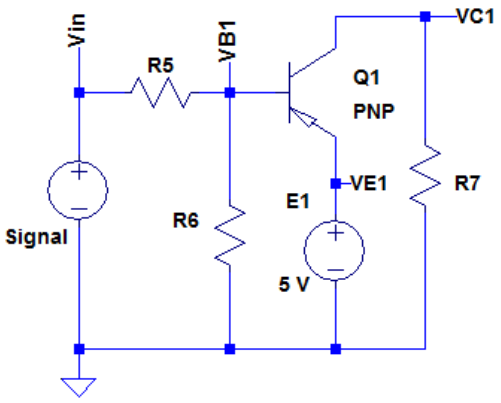


Figura 6.4: Control del V_{in}.

En este caso se utilizó un transistor bipolar pnp como llave. La tensión E_1 se logró mediante un regulador de voltaje 7805, es a partir de tener este valor de tensión en el emisor de Q_1 que se logró tener la señal deseada en el colector a partir del cambio de la señal de entrada.

Para lograr la polarización que lleva al transistor a trabajar en zona de saturación se dimensionaron los valores de R_6 y de R_7 de forma que la corriente de base sea $0{,}1\ mA$ y la corriente de colector de $10\ mA$. Según la hoja de datos del transistor utilizado (BC557C) en saturación con los valores de corriente mencionadas se tiene una caída emisor-base de $0{,}8\ V$ y una tensión emisor-colector de $0{,}2\ V$. Los valores hallados para R_6 y R_7 se presentan en la tabla 6.5.

Estado	$V_{in}(V)$	$V_{EB}(V)$	$V_{EC}(V)$	$I_B(mA)$	$I_C(mA)$	$R_3(k\Omega)$	$R_7(\Omega)$
Saturación (ON)	0	0,8	0,2	0,10	10,0	42	500
Corte (OFF)	5	0	5	0	0	$---$	$---$

Tabla 6.5: Estados de operación de Q_1.

A partir de la tabla 6.5 se puede observar que cuando la entrada es de 5 V se pasa a trabajar en la zona de corte ya que la tensión de base viene dada por la ecuación 6.3.

$$V_{B_1} = \frac{R_6}{R_5 + R_6} * V_{in} \tag{6.3}$$

por lo que eligiendo un valor de $R_5 << R_6$ se tiene que la tensión de base es próxima a 5 V por lo que no circula corriente de base y el transistor permanece en corte. Por lo tanto $R_5 = 1\ k\Omega$.

El vínculo entre las dos secciones de circuitos diseñados y analizados anteriormente se hizo mediante un divisor de tensión tal que la tensión V_2 sea lo más próximo a 5 V cuando el transistor Q_1 este en saturación, valor que se utilizó como referencia para el diseño del circuito de la figura 6.3. En estas condiciones la tensión V_2 viene dada por la ecuación 6.4

$$V_2 = \frac{R_9}{R_8 + R_9} * V_{c_1} \tag{6.4}$$

por lo tanto eligiendo $R_8 << R_9$ se logra el efecto deseado.

Cuando el transistor Q_1 está en corte la tensión V_2 será cero, independiente R_8 y R_9 , por lo tanto el transistor Q_2 también estará en corte al igual que el transistor M1 de acuerdo a lo esperado. La tabla 6.6 resume los resultados obtenidos en el diseño del circuito de control de M1, según el estado de Q_1 y Q_2.

V_{in}	Estado Q_1	$V_{c_1}(V)$	$V_2(V)$	Estado Q_2	$V_{c_2}(V)$	$V_{g_1}(V)$
0	Saturación (ON)	5,0	4,0	Saturación (ON)	0	10,0
5	Corte (OFF)	0	0	Corte (OFF)	12,0	12,0

Tabla 6.6: V_{c_2} y V_{g_1} según el estado de operación de Q_1 y Q_2.

La figura 6.5 muestra el circuito de control del transistor M1 diseñado según lo expuesto.

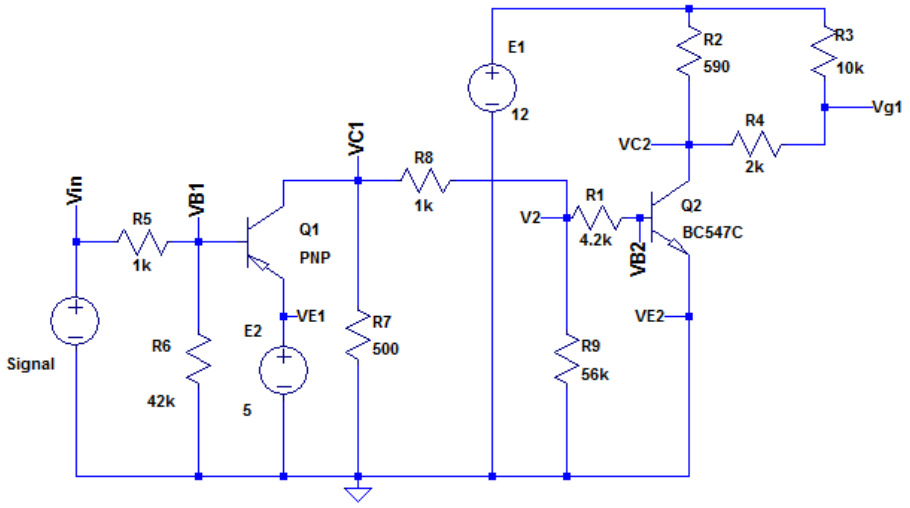


Figura 6.5: Control del V_{g_1}.

A continuación se presentan los resultados de la simulación del circuito diseñado hasta el momento, en la cual se puede observar que la tensión V_{g_1} cumple con los requerimientos impuestos para el diseño.

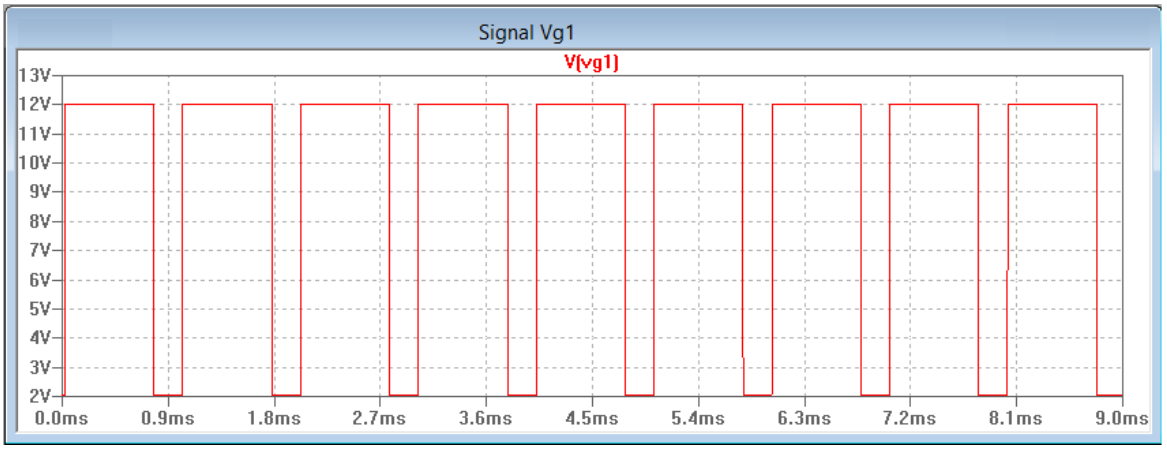


Figura 6.6: Señal de Control del pMos (V_{g_1}).

Una ver diseñado y simulado el circuito de control del transistor M1 se continuó con el diseño del circuito que permite el control del transistor M2. Teniendo en cuenta los resultados obtenidos para la rama de control del transistor M1 se parte de un circuito similar, con la salvedad que la referencia de tensión en este caso es el positivo de la fuente E_3 por lo que la tensión V_{E_3} será igual a $-12\ V$. En la figura 6.7 se presenta el circuito a dimensional para el control de M2.

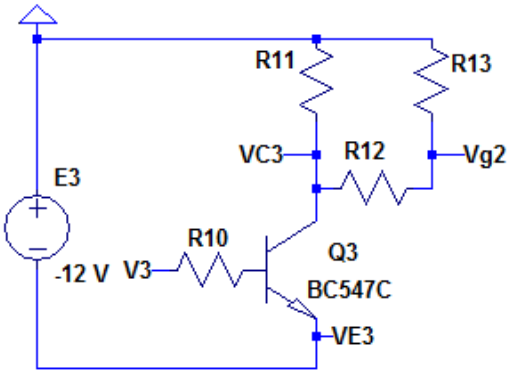


Figura 6.7: Circuito de control del nMos.

En primera instancia se planteó realizar un ajuste de la polarización del transistor Q_3 de forma que opere entre las zonas de corte y saturación. Para ello se analizó la forma de lograr una tensión en V_3 que conmutara entre $-7\ V$ y $-12\ V$ aproximadamente, de esta forma se podría utilizar el circuito de la figura 6.7 con los mismos valores para las resistencias de base y de colector halladas para el circuito de la figura 6.3 y lograr que el transistor Q_3 trabaje entre corte y saturación. Quedando dimensionar los valores de R_{10} y R_{11} para que M2 tenga el comportamiento deseado.

Para esto se plantea un nuevo divisor de tensión conectado entre las tensiones V_{c_1} y V_{E_3}. Según el circuito de la figura 6.8.

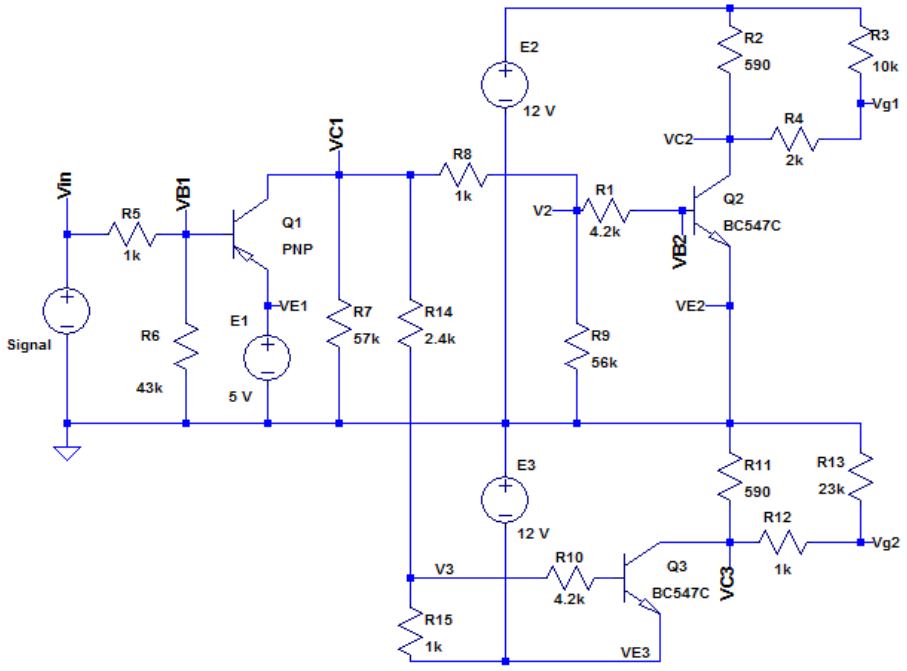


Figura 6.8: Circuito de control de nMos y pMos.

Por lo que si Q_1 está en saturación y se desprecia la corriente de base del transistor Q_3 se tiene la ecuación 6.5.

$$\frac{V_{C_1} - V_3}{R_{14}} = \frac{V_3 - V_{E_1}}{R_{15}} \tag{6.5}$$

De donde se deduce que $R_{14}/R_{15} = 12/5 = 2{,}4$, eligiendo $R_{15} = 1\ k\Omega$ se tiene que $R_{14} = 2{,}4\ k\Omega$. Con los valores de resistencia determinados se tiene que $V_3 = -7\ V$, si Q_1 está en saturación.

Resta analizar que sucede cuando Q_1 está en corte, en este caso V_3 viene dada por la ecuación 6.6.

$$V_3 = \frac{R_7 + R_{14}}{R_7 + R_{14} + R_{15}} * V_{E_3} \tag{6.6}$$

por lo tanto, si se quiere lograr una tensión $V_3 \approx 12\ V$ se debe ajustar el valor de R_7. Despejando para R_7 y tomando $V_3 = -11{,}8\ V$ se tiene la ecuación 6.7.

$$R_7 = \frac{(R_{14} + R_{15})V_3 - R_{14}V_{E_3}}{V_{E_3} - V_3} \tag{6.7}$$

Para determinar el valor de la resistencia $R_7 \approx 56,5\ k\Omega$ se despreció el efecto de las resistencias R_8 y R_9 por lo que es necesario hallar el valor de las tensiones V_{C_1} y V_3 para verificar si cumple con el requerimiento planteado, ya que el valor obtenido para R_7 es de igual magnitud que $R_8 + R_9$. Por lo que V_{C_1} viene dada por la ecuación 6.8.

$$V_{C_1} = \frac{R_{eq}}{R_{eq} + R_{14} + R_{15}} * V_{E_3} \tag{6.8}$$

siendo $R_{eq} = R_7//R_8 + R_9 = 27{,}5\ k\Omega$, el resultado anterior se obtuvo tomando $R_7 = 57\ k\Omega$. Si se desprecia $R_8 + R_9$ se tiene que $V_{C_1} = -11{,}3\ V$ por lo que se consideró que el haber despreciado las resistencias mencionadas no afectó significativamente el resultado esperado. Haciendo un cálculo similar se tiene que $V_3 = -11{,}6\ V$ si se consideran las resistencias R_8 y R_9, en caso contrario $V_3 = -11{,}8\ V$.

El haber conectado este nuevo divisor resistivo al colector de Q_1 hace que la tensión V_{C_1} ya no sea cero cuando Q_1 esté en corte, sino que vale $-10{,}7\ V$. Vale observar que este valor de tensión negativa no afecta el comportamiento del transistor Q_2 por lo cual puede ser aceptado, de esta forma la corriente de emisor-colector de Q_1 pasa a ser 877 μA en lugar de 10 mA como se había establecido anteriormente.

Con los valores obtenidos para R_7 , R_{14} y R_{15} se tiene que Q_3 conmuta entre las zonas de corte y saturación como se espera, esto se puede observar en la figura 6.9 donde se muestra como varia la tensión V_{C_3} entre 0 V y $-11{,}9\ V$.

Con los valores obtenidos para V_{C_3} resta ajustar los valores de tensión V_{g_2} adecuados para controlar M2. Según lo expuesto en la tabla 6.3 para que M2 este en corte se debe verificar que $V_{gs} \leq V_{t_o} = 1{,}8\ V$. Por lo tanto se determinó los valores de R_{12} y R_{13} para que $V_{g_2} = -11{,}5\ V$ cuando $V_{C_3} = -11{,}9\ V$. La relación entre ambas tensiones viene dada por la ecuación 6.9.

$$V_{g_2} = \frac{R_{12}}{R_{12} + R_{13}} * V_{C_3} \tag{6.9}$$

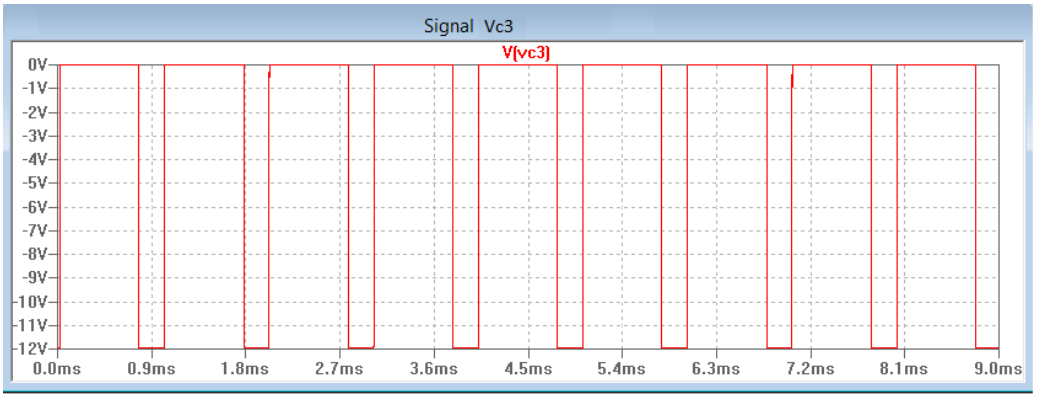


Figura 6.9: Señal en el colector del transistor Q_3.

tomando $R_{12} = 1\ k\Omega$ se tiene que $R_{13} = 23\ k\Omega$, en estas condiciones se tiene que $V_{gs} = 0{,}5\ V$ verificándose la condicione requerida. La otra condición para que M2 este en corte se cumple naturalmente, ya que cuando M2 está en corte M1 está en saturación por lo que la tensión de drain es 12 V y $V_{ds} = 24\ V > 0\ V$.

Resta verificar que sucede cuando en la tensión $V_{C_3} = 0\ V$, en estas condiciones se tiene que V_{g_2} también es cero, por lo que se verifica de inmediato que $V_{gs} = 12\ V > V_{t_o} = 1{,}8\ V$. Para comprobar que la otra condición para que M2 este en saturación se verifica se utiliza un argumento similar al utilizado cuando se analizó el estado de saturación de M1. Dado que la corriente máxima que requiere la carga es de 36 mA y que la $R_{ds_{on}} = 0{,}15\ \Omega$ se tiene que la caída de tensión drain-source verifica que $0 < V_{ds} = 5{,}4\ mV < V_{gs} - V_{t_0} = 10{,}2\ V$.

En la tabla 6.7 se resumen los resultados obtenidos en el diseño del circuito de control del M2, según el estado de Q_1 y Q_3.

V_{in}	Estado Q_1	$V_{c_1}(V)$	$V_3(V)$	Estado Q_3	$V_{c_3}(V)$	$V_{g_2}(V)$
0	Saturación (ON)	5,0	$-7{,}0$	Saturación (ON)	$-11{,}9$	$-11{,}5$
5	Corte (OFF)	$-11{,}0$	$-11{,}8$	Corte (OFF)	0	0

Tabla 6.7: V_{c_3} y V_{g_2} según el estado de operación de Q_1 y Q_3.

En la figura 6.10 se muestra el resultado obtenido al simular el circuito diseñado tomando la tensión de control de M2. Como se puede observar la misma varía entre los valores buscados con el diseño, esto es, entre 0 V y $-11{,}5$ V.

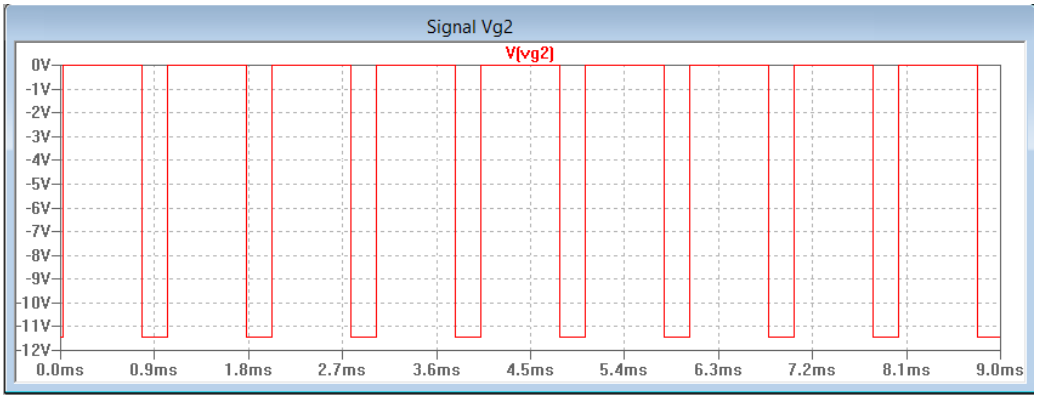


Figura 6.10: Señal en el colector del transistor M2.

Para verificar el funcionamiento del circuito diseñado se implementó el modelo LTSpice de los transistores nMos y pMos con que se cuenta y se realizó la simulación del circuito mostrado en la figura 6.1 al inicio del presente capitulo obteniendo-se el resultado presentado en la figura 6.11.

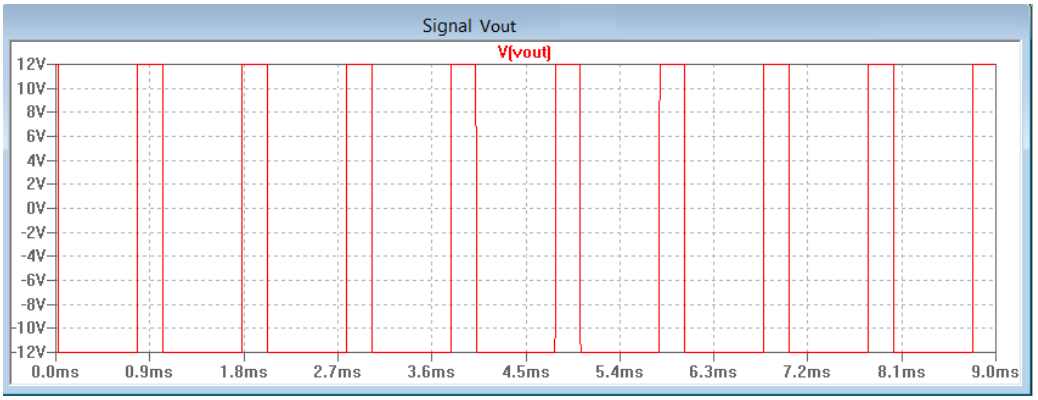


Figura 6.11: Señal a la salida del circuito de la 6.1.

Como se puede ver la señal de salida cumple con las especificaciones del diseño en lo que refiere a que es una señal PWM de amplitudes ± 12 V. A continuación en la figura 6.12 se presenta las señales PWM de entrada y de salida superpuesta para verificar que los 12 V a la salida se obtiene cuando la entrada es 0 V y que la salida es -12 V cuando la entrada es 5 V.

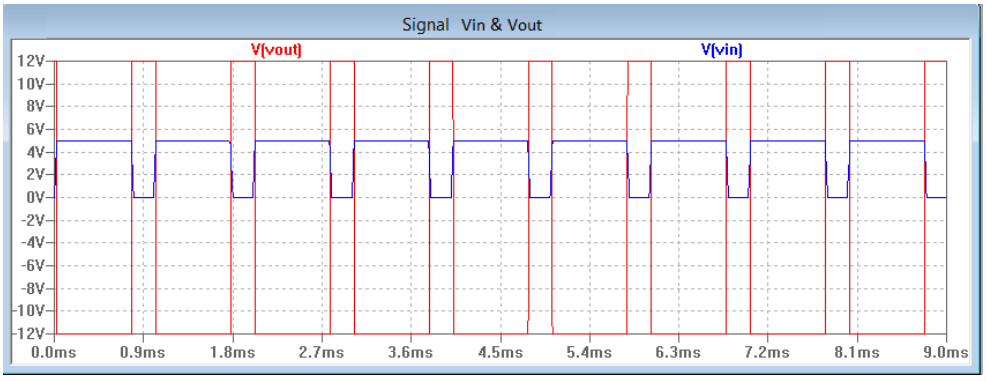
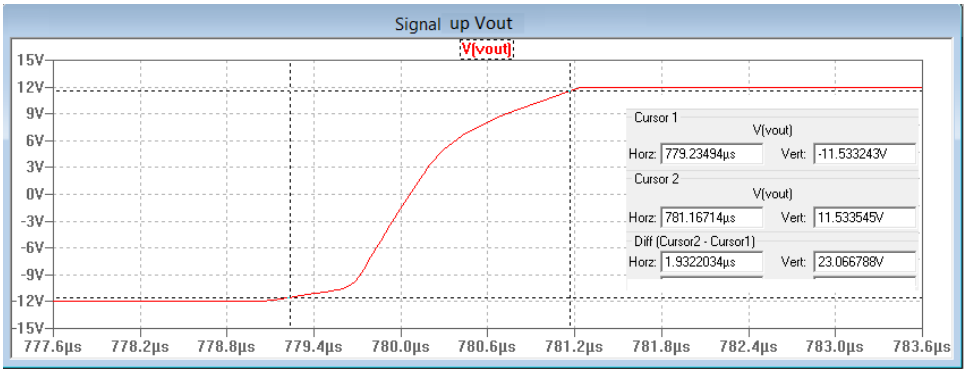
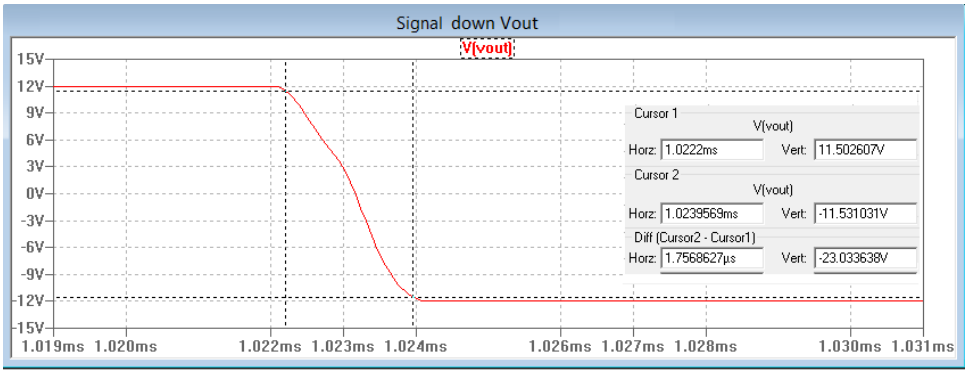


Figura 6.12: Señales de entrada y salida en el circuito de la figura 6.1.

Si bien no se presenta en las gráficas de las figuras precedentes también se verificó que la frecuencia de la señal de salida sea 1 kHz. Resta verificar que se está cumpliendo con los tiempos máximos de subida y bajada de la señal. Como se puede observar las figuras 6.13a el tiempo de subida es de 1,93 μs, mientras que el de bajada es 1,76 μs como se ve en la figura 6.13b.



(a) Tiempo de Subida.



(b) Tiempo de Bajada.

Figura 6.13: Tiempo de Subida y de Bajada de la señal de salida.

6.4. Armado y Verificación del Circuito.

Una vez diseñado y simulado el circuito de la figura 6.1 se procedió a armar una versión de prueba en una protoboard. En esta sección se presentan los resultados obtenidos al realizar medidas de las señales de control en los distintos puntos de interés.

En primera instancia se presenta la señal de entrada para verificar que es una señal PWM de 0 V a 5 V con una frecuencia de 1 kHz.

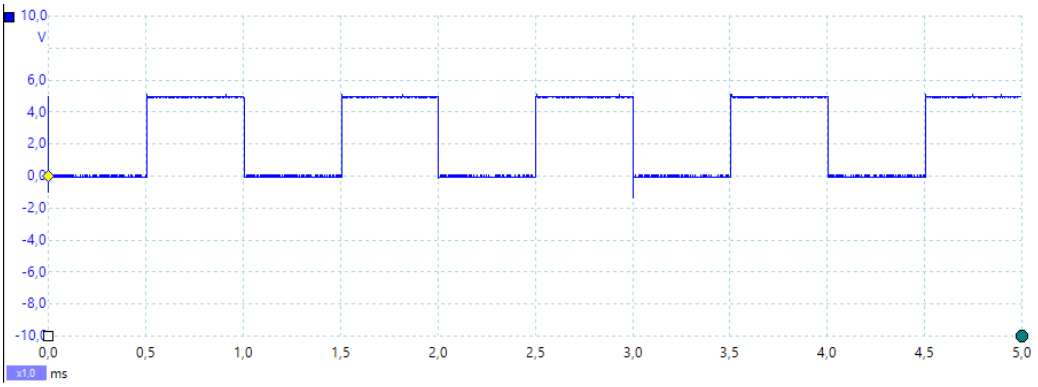


Figura 6.14: Señal PWM generada por el Arduino.

Como se puede ver en la figura 6.14 la señal PWM generada por el Arduino tiene una amplitud de 5 V con un periodo de 1 ms por lo que cumple con los requerimientos necesario.

Siguiendo con el análisis del circuito implementado se presenta en la figura 6.15 la señal de salida obtenida en vacío, en la misma se puede ver que cumple con los requerimientos de amplitud y frecuencia requerido, sin embargo presenta un sobretiro indeseado durante la subida de la señal por lo que se debió realizar una análisis de las tensiones de interés para así determinar la causa del mencionado sobretiro.

Figura 6.15: Señal PWM a la salida.

En la figura 6.16 se muestra una imagen ampliada del sobretiro que presenta la señal de salida.

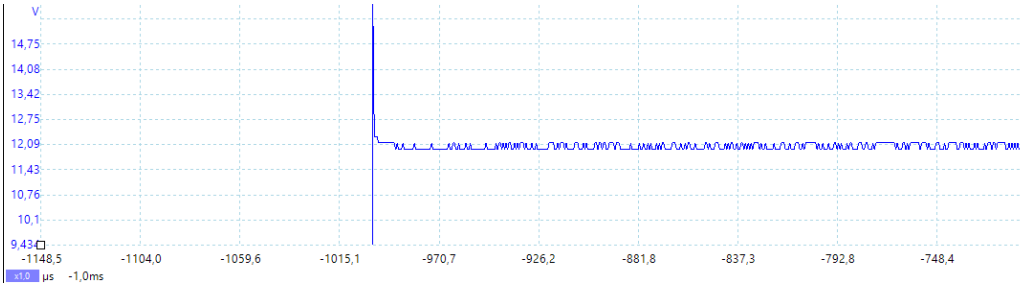
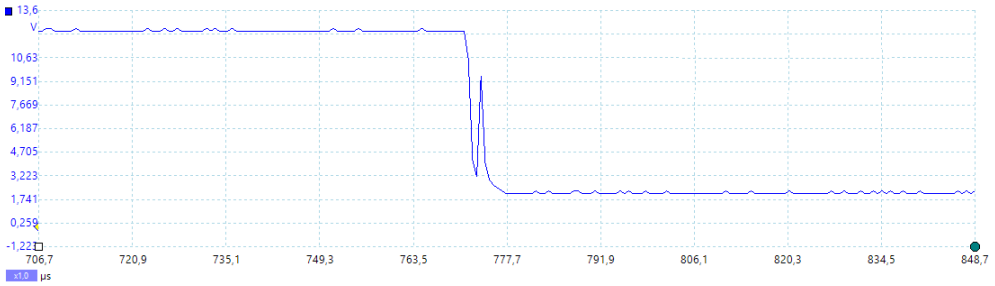
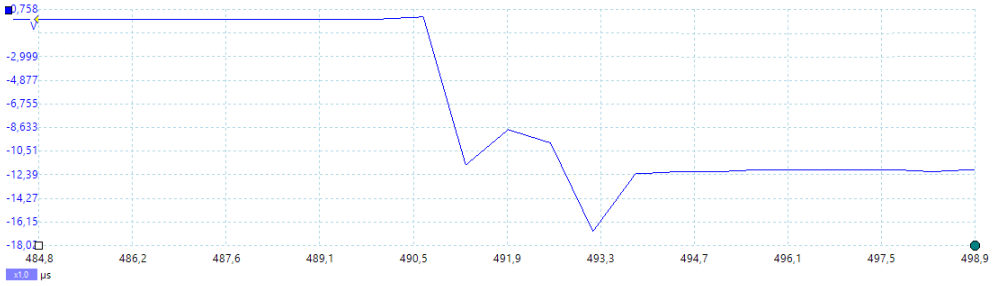


Figura 6.16: Sobre-tiro en la subida de la señal PWM a la salida.

Para resolver el problema que presenta la señal de salida se realizaron medidas en distintos puntos del circuito, observandose que las señales V_{g1} y V_{C3} presentan una discontinuidad durante la bajada como se puede observar en la figuras 6.17a y 6.17b.



(a) Discontinuidad en la bajada de V_{g1}.



(b) Discontinuidad en la bajada de V_{C3}.

Figura 6.17: Discontinuidad de las señales V_{g1} y V_{C3}.

Una vez realizada las medidas de tensión en los distintos punto del circuito encontrándose las distorsiones mencionadas previamente se buscaron distintas alternativas para resolverlo.

En primera instancia se modificó el punto de operación del transistor Q_3 haciendo que pase a operar a un valor de corriente de base de 6,6 μA mediante una resistencia de base de 75 $k\Omega$ y con una corriente de colector de 7,5 mA usando una resistencia de colector de 1,6 $k\Omega$. Con esta modificación se logró resolver la discontinuidad en V_{C3} aunque la señal del salida no se vio modificada.

Entonces se pasó a modificar la resistencia de base del transistor Q_1 utilizando una del mismo valor que el usado para Q_3, también se utilizó la misma resistencia de colector. Con

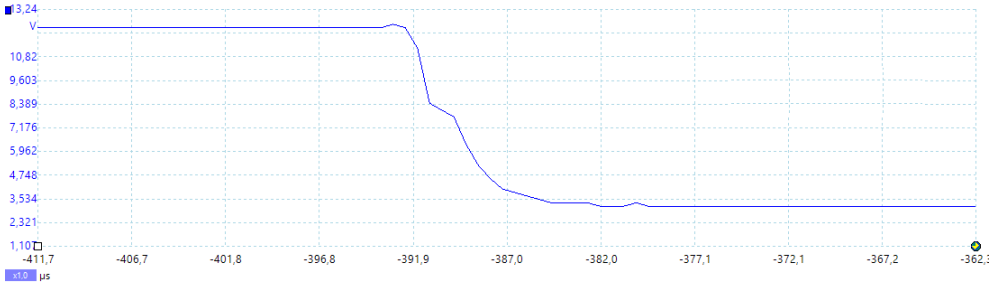
estas modificaciones no se obtuvo cambio en la señal de salida pero si disminuyo la distorsión en V_{g1}.

A partir de los resultados obtenidos se probaron dos modificación simultaneas para modificar el ajuste de tensón V_{g1}, se remplazaron las resistencia R_2 y R_4 por dos resistencia variable ajustadas en los valores de las resistencia que se tenía y se modificaron sus valores viendo como afectaban estas a la señal V_{g1}.

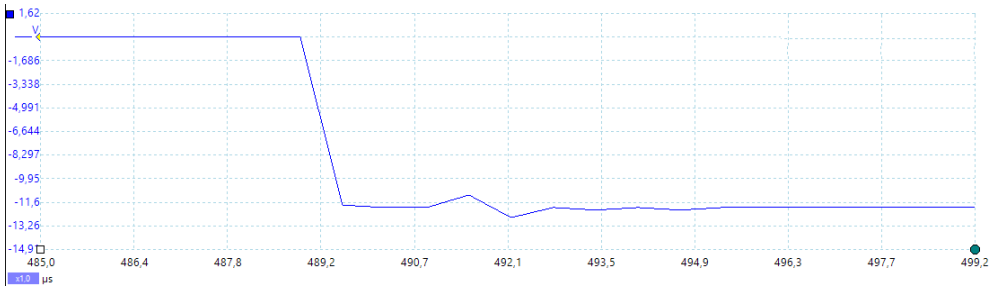
Se observó que, si R_2 disminuía y R_4 aumentaba, la distorsión de V_{g1} se atenuaba. De ahí se pasó a ver qué sucedía con la señal de salida y se observó que el sobretiro también disminuía con la misma variación de las resistencias mencionadas.

Se realizó el ajuste hasta que la señal de salida no presentara sobretiro. Una vez ajustados los valores de las dos resistencia se medió el valor de cada una de ellas y se las remplazó por valores próximos a los obtenidos, para R_2 se utilizo una de 1 $k\Omega$ y para R_4 una de 3,3 $k\Omega$.

Una vez hechas las modificaciones mencionadas se tomaron nuevamente las medidas de las tensiones V_{g1} y V_{C3} obteniéndose los resultados mostrados en las figuras 6.18a y 6.18b.



(a) Bajada de V_{g1} sin distorsión.



(b) Bajada de V_{C3} sin distorsión.

Figura 6.18: Señales V_{g1} y V_{C3} sin distorsión.

Como se puede observar con las modificaciones realizadas al circuito, se logró mejorar la distorsión que presentaban ambas señales analizadas durante la bajada.

A continuación se presenta en la figura 6.19 la señal de salida luego de los ajustes realizados. Como se puede observar la misma ya no presenta más el sobretiro.

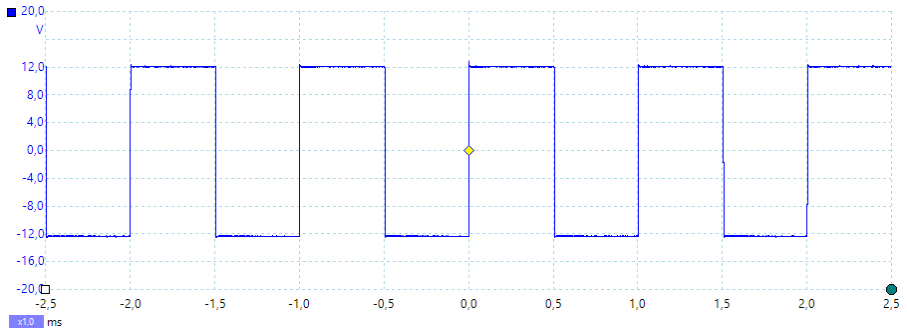
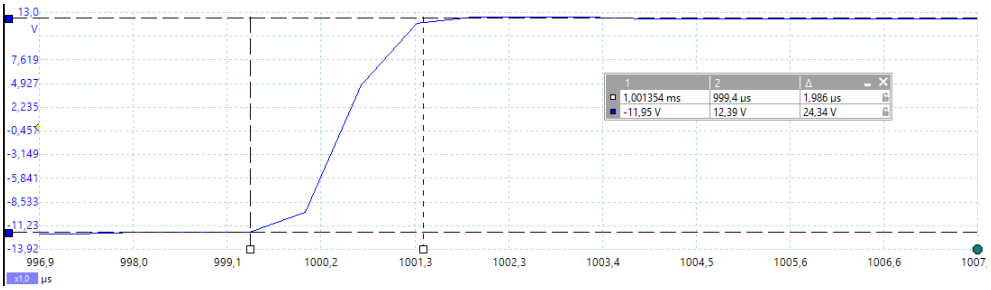
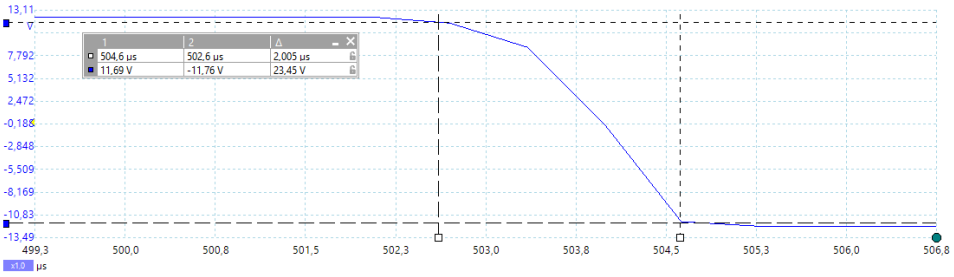


Figura 6.19: Señal de Salida.

Resta verificar si se cumplen lo referidos a los tiempos de subida y bajada. En las figuras 6.20a y 6.20b se muestra las medidas del tiempo de subida y bajada respectivamente, como se puede ver se tiene 1,97 μs para la subida y 2,01 μs en la bajada. Vale observar que el tiempo fue medido entre el máximo y el mínimo de la señal por lo que verifica lo requerido por la norma, ya que éste tiempo debe ser menor a 2 μs entre el 10 % y el 90 % del rango de variación de la señal.



(a) Tiempo de Subida.



(b) Tiempo de Bajada.

Figura 6.20: Tiempo de Subida y de Bajada a la salida en vacío.

De esta forma queda verificado el correcto funcionamiento del circuito desarrollada según los criterios presentados, por lo que podrá ser implementado en circuito impreso.

Vale aclarar que además de los valores de las resistencias modificados para lograr el ajuste de las señales analizadas previamente se utilizaron algunas valores de resistencias que difieren sensiblemente de los valores definidos cuando se diseñó el circuito. Estas diferencias se deben a la disponibilidad de resistencias con que se contaba al momento de armar el circuito. Los valores que finalmente se utilizaron se presentan en la próxima sección.

Todo el análisis presentado previamente hace referencia al circuito diseñado para la comunicación, mediante la conexión del hilo de control, entre el SAVE y el VE. Resta decir que a la

salida del puente formado por los transistores mosfet se usó una resistencia de $1k\Omega$, conectada entre dicho punto y el neutro del circuito, necesaria para el correcto funcionamiento del puente. También a la salida se conectó otra resistencia de 1 $k\Omega$ prevista por el protocolo IEC para lograr la comunicación entre el SAVE y el VE a través del hilo de control.

Además de las dos resistencias mencionadas fue necesario conectar un divisor resistivo luego de la ultima resistencia de 1 $k\Omega$ mencionada para poder realizar la medida de tensión, desde el SAVE, según el estado de conexión o de carga del vehículo. A dicho divisor también se le conectó un diodo para evitar tener tensiones negativas ya que las medidas realizadas mediante las entradas analógicas del Arduino deben estar entre 0 V y 5 V.

Por otra parte se implementó la comunicación entre el SAVE y el VE mediante el hilo de proximidad, necesario para determinar si el conector es adecuado para la carga y que corriente máxima soporta el cable. Esta se realizó mediante una resistencia de 220 Ω con un extremo conectado a 5 V y el otro al hilo de proximidad del conector.

6.5. Implementación en Circuito Impreso.

En la presente sección se presentará el proceso seguido en la implementación del circuito desarrollado en PCB. Para diseñar la PCB se utilizó el servicio web de desarrollo EasyEDA [2] que permite el diseño y la simulación del circuito. A partir del cual se edito el circuito de la figura 6.21.

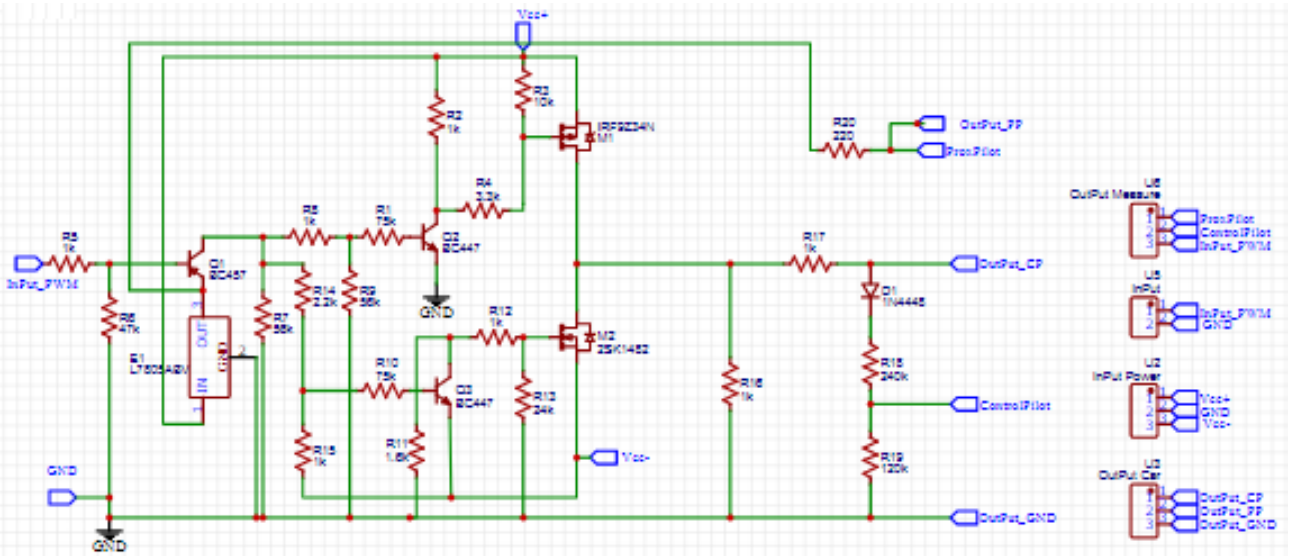


Figura 6.21: Circuito editado para obtener el PCB.

Como resultado se obtuvo el diseño de las pistas en dos capas mostrada en la figura 6.22.

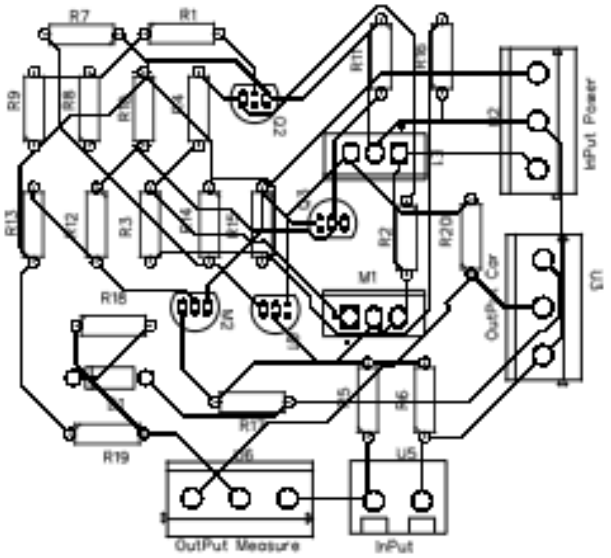
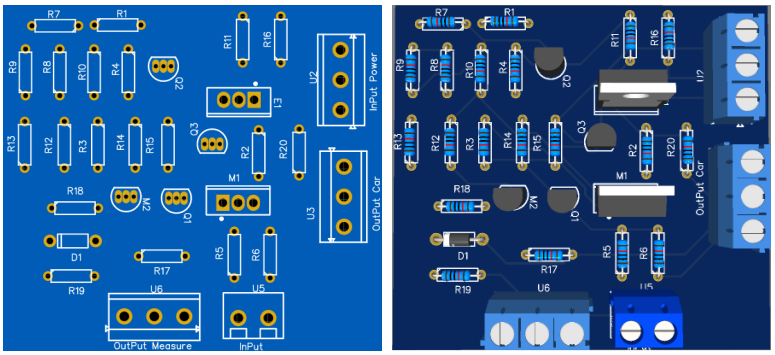


Figura 6.22: Circuito impreso en dos capas.

A partir del circuito mostrada en la figura 6.22 se obtuvo una vista del plano de la placa impresa que se muestra en la figura 6.23a y un modelo 3D mostrado en la figura 6.23b.

[2] https://easyeda.com/es



(a) PCB. (b) PCB 3D.

Figura 6.23: Simulación del PCD y PCB 3D.

Dado que el costo de implementación un sola placa de circuito impreso no justificaba su realización se opto por implementar una primer versión de la misma con una placa perforada. Para ello se tuvo en cuenta todo lo descripto en el proceso de diseño, lográndose el circuito mostrado en la figura 6.24.

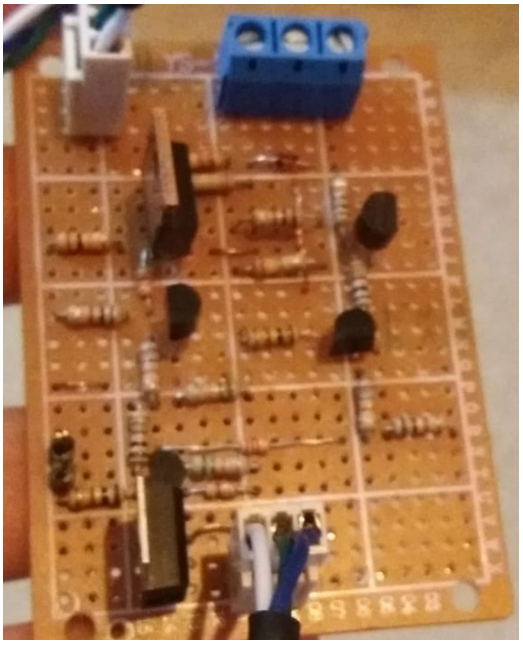


Figura 6.24: Circuito implementado.

Todos los componentes del circuito implementado se pueden encontrar en el mercado local y tiene un costo total aproximado de 750 pesos uruguayos.

En la tabla 6.8 se presenta la lista de materiales utilizados en el circuito de la figura 6.24.

ID	Nombre	Descripción
1	$47k\Omega$	R_6
2	$1k\Omega$	R_2,R_5,R_8,R_{12},R_{15},R_{16} R_{17}
3	$75k\Omega$	R_1,R_{10}
4	$2,2k\Omega$	R_{14}
5	$3{,}3k\Omega$	R_4
6	$10k\Omega$	R_3
7	$24k\Omega$	R_{13}
8	220Ω	R_{20}
9	$240k\Omega$	R_{18}
10	$120k\Omega$	R_{19}
11	$1{,}6k\Omega$	R_{11}
12	$56k\Omega$	R_7, R_9
13	$1N4448$	D_1
14	$BC457$	Q_1
15	$BC447$	Q_2,Q_3
16	$IRF9Z34N$	M_1
17	$2SK1482$	M_2
18	$L7805$	E_1
19	$ConectorconTornillosde3P$	Dos
20	$ConectordePinesde3P$	Uno
21	$ConectordePinesde2P$	Uno

Tabla 6.8: Componentes del circuito IEC.

CAPÍTULO 7

CÓDIGO IMPLEMENTADO SEGÚN PROTOCOLO IEC.

7.1. Descripción del Código Implementado en Arduino.

En la esta sección se presenta la descripción del código implementado para realizar la comunicación entre el SAVE y el VE. En la tabla 7.1 se presentan los estados de la comunicación, la descripción de los mismos, así como los estados a los que puede ocurrir una transición.

Estado	Descripción	Estado Siguiente
A0	Fuera de servicio.	**A1** - A6
A1	Cargador listo, esperando tarjeta.	**A2** - A0 - **A6**
A2	Esperando validación de la tarjeta.	**A3** - A0 - A1 - A4 - **A5**
A3	Se verifica si el vehículo está conectado.	**A4** - A0 - **A6**
A4	Cargando vehículo.	**A2** - A0 - A1
A5	Sesión cerrada, esperar desconexión del vehículo.	**A1** - A0
A6	Error en el conector. Conector no válido.	A0 - A1

Tabla 7.1: Estados del Arduino.

El código implementado para cumplir con lo mencionado se presenta en el Anexo C. Y los archivos correspondientes se pueden encontrar en el soporte digital que acompaña este documente en la carpeta Arduino.

- El Arduino permanece en el estado **A0** siempre que la Raspberry pierda comunicación con el servidor o que dicha comunicación no se halla inicializado aún.
- En el estado **A1**, el SAVE está listo para ser usado y el Arduino se encuentra esperando que un usuario pase una tarjeta. El Arduino no verifica la validez de la tarjeta, sino que solo le envía a la Raspberry el código identificador leído, y espera que la Raspberry le responda si es una tarjeta válida o no lo es.
- El Arduino permanecerá en el estado **A2** hasta que reciba la notificación de la Raspberry que le permite saber si la tarjeta es válida (o no) para realizar la operación deseada, que puede ser iniciar o parar una sesión de carga.
- Luego de iniciada una sesión de carga, en el estado **A3** se verifica que el vehículo esté conectado. Y en caso de que lo esté, se pasa de inmediato al estado de carga **A4**; de lo contrario, se espera que el usuario conecte el vehículo.
- Se permanece en el estado **A4** durante la sesión de carga del vehículo. Y la misma terminará si se cierra la sesión de carga (mediante el uso de la misma tarjeta que la inició) o si se produce una desconexión intempestiva del vehículo o un error de conexión con el servidor.
- Una vez cerrada la sesión de carga se permanece en el estado **A5** hasta que el vehículo sea desconectado u ocurra un error de conexión con el servidor.
- En el estado **A6** se permanece siempre que se detecte un error en el conector que solo se restablece al ser desconectado o si ocurre un error de conexión con el servidor y luego de un tiempo se pasa al estado fuera de servicio.

En el diagrama de estado de la figura 7.1 se representa de forma gráfica las transiciones de estado y en la tabla 7.2 se exponen las condiciones requeridas para realizar cada una de las transiciones. Las transiciones consideradas exitosas son marcadas en color verde, en la figura 7.1, para mayor comprensión. Y una secuencia de exitosa tendría el siguiente orden de transición:
A0 $\rightarrow$ **A1** $\rightarrow$ **A2** $\rightarrow$ **A3** $\rightarrow$ **A4** $\rightarrow$ **A2** $\rightarrow$ **A5** $\rightarrow$ **A1**

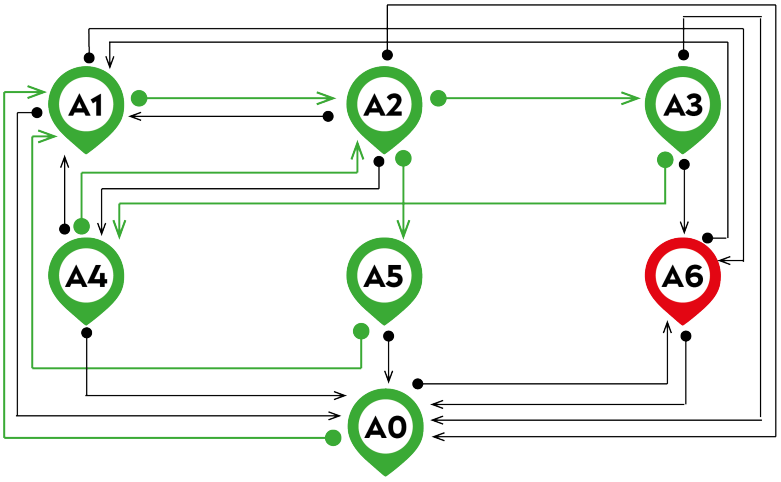


Figura 7.1: Diagrama de estados del Arduino.

Inicial	Final	Condición de transición
A0	**A1**	Si se inicia correctamente la comunicación con el servidor y no hay error en la comunicación con el vehículo.
A0	**A6**	Cuando se inicia correctamente la comunicación con el servidor y hay error en la comunicación con el vehículo.
A1	A0	Si ocurre un error en la comunicación con el servidor.
A1	**A2**	Si se lee una tarjeta.
A1	**A6**	Si hay error en la comunicación con el vehículo.
A2	A0	Si ocurre un error en la comunicación con el servidor.
A2	A1	Si la tarjeta leída no es validada por el centro de control de carga para iniciar la carga.
A2	**A3**	Si el servidor autoriza la carga.
A2	A4	Si la tarjeta leída no es validada por el centro de control de carga para detener la carga.
A2	**A5**	Si el servidor autoriza detener una sesión de carga.
A3	A0	Si ocurre un error en la comunicación con el servidor.
A3	**A4**	Si se verifica que el vehículo está correctamente conectado.
A3	**A6**	Si hay error en la comunicación con el vehículo.
A4	A0	Si ocurre un error en la comunicación con el servidor.
A4	A1	Si se desconecta el vehículo sin cerrar la sesión de carga con una tarjeta valida.
A4	**A2**	Si se lee una tarjeta.
A5	A0	Si ocurre un error en la comunicación con el servidor.
A5	**A1**	Si se desconecta el vehículo luego de cerrar correctamente una sesión de carga.
A6	A0	Si ocurre un error en la comunicación con el servidor.
A6	A1	Si se recupera la comunicación con el vehículo.

Tabla 7.2: Transiciones de estado del Arduino.

7.1.1. Estado A0: Fuera de Servicio.

La primera condición que se verifica, es si la Raspberry envió un código de inicio (un uno) que en código ASCCI sería 49.

Mientras no llegue un código 49, el Arduino se mantendrá en el estado A_0, mostrando en el Display el mensaje de "FUERA DE SERVICIO".

Cuando el código 49 sea recibido, se verifica si el voltaje del Control Pilot, está en el entorno de 12 V (entre 11,5 V y 12,5 V).

- Si está en 12 V, entonces se pasa al siguiente estado A_1.
- Si el voltaje esta fuera de ese entorno, se pasa al estado error A_6, se le envía un código error a la Raspberry y en el Display se muestra el mensaje: "ERROR EN CONECTOR".

7.1.2. Estado A1: Cargador Listo, Esperando Tarjeta.

La primera condición que se verifica, es si la Raspberry envió un código error (un cero) que en código ASCCI sería 48. Cuando el código 48 sea recibido, se resetea el pin que genera el PWM, se espera un instante y se abre el contactor. Luego se pasa al estado inicial A_0.

Mientras no llegue un código 48, el Arduino verifica si el voltaje del Control Pilot, está en el entorno de 12 V (entre 11,5 V y 12,5 V) o está en el entorno de de 9 V (entre 8,5 V y 9,5 V):

- Si el voltaje está fuera de cualquiera de esos entornos, se pasa al estado error A_6, se le envía un código error a la Raspberry y en el Display se muestra el mensaje: "ERROR EN CONECTOR".
- Si el voltaje está dentro de cualquiera de esos entornos: el siguiente paso es chequear si se leyó alguna tarjeta.

  Mientras no se lea ninguna tarjeta, el Arduino se mantendrá en el estado A_1 y en el Display se muestran los mensajes: "INICIE SESION, PASE TARJETA" y "BIENVENIDO GMH" alternativamente.

  Cuando en el estado A_1 se lea una tarjeta (***CardValue == true***), se pasará al siguiente estado A_2 y se guarda el instante en que fue leída la tarjeta, en la variable **TimeCard**.

7.1.3. Estado A2: Esperando Validación de Tarjeta.

En este estado, se espera el código enviado por la Raspberry, que notificará al Arduino si la tarjeta es válida (o no) para iniciar o detener una sesión de carga. A este estado, se llega desde el estado A_1 o desde el estado A_4. Si la tarjeta resulta ser válida, entonces se iniciará una sesión de carga (si el Arduino venía del estado A_1) o se detendrá la sesión de carga (si el Arduino venía del estado A_4).

Lo primero que se chequea en el estado A_2 es si se excedió el tiempo máximo de espera de la respuesta. Cuando eso acontezca, el Arduino envía el código error a la Raspberry, se resetea el pin que genera el PWM, se espera un instante y se abre el contactor. Luego se pasa al estado de error A_6 y se muestra el mensaje "FUERA DE SERVICIO".

Mientras no se exceda ese tiempo máximo, se chequea si hubo un código enviado por la Raspberry, el cual será guardado en la variable **StatusCard**.

- Si **StatusCard** es cero (en código ASCCI sería 48), entonces significa que hubo un error de comunicación entre el servidor y la Raspberry. Así que se pasará al estado A_4 para chequear si el error se mantiene.
- Si **StatusCard** es un dos (en ASCCI sería 50), entonces la tarjeta fue etiquetada como **válida** para **iniciar** la carga[1]. En dicho caso, se pasa al siguiente estado A_3.
- Si **StatusCard** es un cuatro (en ASCCI sería 52), entonces la tarjeta fue etiquetada como **inválida** para **iniciar** la carga. En dicho caso, se vuelve al estado anterior A_1 a esperar una nueva tarjeta para iniciar la carga y se muestra en el Display el mensaje "TARJETA RECHAZADA, PASE OTRA TARJETA".
- Si **StatusCard** es un cinco (en ASCCI sería 53), entonces la tarjeta fue etiquetada como **válida** para **iniciar** la carga pero la sesión de carga fue **rechazada** por el servidor. En dicho caso, se pasa al estado A_5 y se muestra en el Display el mensaje "CARGA RECHAZADA, DESC EL VEHICULO".
- Si **StatusCard** es un tres (en ASCCI sería 51), entonces la tarjeta fue etiquetada como **válida** para **detener** la carga[2]. En dicho caso, se detiene el PWM, se espera un instante y se abre el contactor. Luego se pasa al estado A_5.
- Si **StatusCard** es un seis (en ASCCI sería 54), entonces la tarjeta fue etiquetada como **inválida** para **detener** la carga. En dicho caso, se pasa al estado A_4 y se muestra en el Display el mensaje "TARJETA RECHAZADA, PASE OTRA TARJETA".

Mientras no se lea ninguna tarjeta, el Arduino se mantendrá en el estado A_1 y en el Display se muestran los mensajes: "TARJETA LEIDA" y "PROCESANDO LA SOLICITUD AGUARDE" alternativamente.

7.1.4. Estado A3: Se Verifica si el Vehículo está Conectado.

La primera condición que se verifica, es si la Raspberry envió un código error (un cero) que en código ASCCI sería 48. Cuando el código 48 sea recibido, se pasa al estado inicial A_0.

Mientras no llegue un código 48, el Arduino verifica si el voltaje del Control Pilot, está en el entorno de 9 V (entre 8,5 V y 9,5 V):

Si está fuera de dicho entorno de 9 V, entonces el Arduino se mantiene en el mismo estado, pero muestra en el Display el mensaje "CONECTAR EL VEHICULO". Mientras esté dentro del entorno de 9 V (con el vehículo conectado) se chequea cuánto amperaje admite el cable del conector detectado y con eso se calcula el DutyCicle del PWM a generar.

- Si el **Cicle** es igual a *CodigoError*, entonces se pasa al estado error A_6, se le envía un código error a la Raspberry y en el Display se muestra el mensaje: "ERROR EN CONECTOR".
- Si el **Cicle** es distinto a *CodigoError*, entonces primero se cierra el contactor, se espera un instante y se comienza a generar el PWM con un *DutyCicle* = ***Cicle***. Por último se pasa al siguiente estado A_4.

[1] El código ASCCI 50 sólo se puede recibir de la Raspberry, si el estado anterior del Arduino era el A_1.
[2] El código ASCCI 51 sólo se puede recibir de la Raspberry, si el estado anterior del Arduino era el A_4.

7.1.5. Estado A4: Cargando Vehículo.

En este estado, el objetivo es mantener el vehículo cargando, hasta que suceda alguno de los siguientes tres eventos:

Se pasa una tarjeta para detener la sesión, se desconecta el vehículo o hay un código error enviado por Raspberry por problemas de comunicación con el Servidor de UTE.

En el código, la primera condición que se verifica, es si la Raspberry envió un código error (un cero) que en código ASCCI sería 48.

Cuando el código 48 sea recibido, se resetea el pin que genera el PWM, se espera un instante y se abre el contactor. Luego se pasa al estado inicial A_0.

Mientras no llegue un código 48, el Arduino verifica si el voltaje del Control Pilot, está en el entorno de 12 V (entre 11,5 V y 12,5 V):

- Si el voltaje está dentro de ese entorno, entonces el vehículo se desconectó. En dicho caso: primero se detiene el PWM, se espera un instante y luego se abre el contactor. Luego se pasa al estado A_1.
- Si el voltaje está fuera de ese entorno, entonces el vehículo se seguirá cargando, hasta que se pase la misma tarjeta que inicio la sesión de carga y el centro de control de carga autorice detener la sesión de carga.

  Mientras no haya una lectura de tarjeta, se mostrarán en el Display los mensajes "VEHICULO CARGANDO" y "NO DESCONECTAR EL VEHICULO", alternativamente.

  Cuando en el estado A_4 se lea una tarjeta (***CardValue*** $^{--}$ **true**), se pasará al estado A_2 y se guarda el instante en que fue leída la tarjeta, en la variable **TimeCard**.

7.1.6. Estado A5: Sesión Cerrada, Esperar Desconexión del VE.

La primera condición que se verifica, es si la Raspberry envió un código error (un cero) que en código ASCCI sería 48.

Cuando el código 48 sea recibido, se resetea el pin que genera el PWM, se espera un instante y se abre el contactor. Luego se pasa al estado inicial A_0.

Mientras no llegue un código 48, el Arduino verifica si el voltaje del Control Pilot, está en el entorno de 12 V (entre 11,5 V y 12,5 V):

- Si está fuera del entorno de 12V, entonces el Arduino se mantiene en el mismo estado, pero muestra en el Display el mensaje "CARGA FINALIZADA DESC. EL VEHICULO"
- Si el voltaje está dentro del entorno de 12 V, el vehículo ya se desconecto, así que se pasa al estado A_1.

7.1.7. Estado A6: Error en el Conector.

La primera condición que se verifica, es si la Raspberry envió un código error (un cero) que en código ASCCI sería 48. Cuando el código 48 sea recibido, se resetea el pin que genera el PWM, se espera un instante y se abre el contactor. Luego se pasa al estado inicial A_0.

Mientras no llegue un código 48, el Arduino verifica si el voltaje del Control Pilot, está en el entorno de 12 V (entre 11,5 V y 12,5 V):

- Si está fuera del entorno de 12 V, entonces el Arduino se mantiene en el mismo estado. Y luego de un intervalo de tiempo mayor a **DelayError**, se envía un código error a la Raspberry.

- Si el voltaje está dentro del entorno de 12 V, el error de conexión con el vehículo se despejó y se pasa al estado A_1 para poder iniciar una nueva sesión. También se le envía un código de reinicio a la Raspberry, para notificarle que el problema fue resuelto.

7.1.8. Otras Funciones Utilizadas Durante la Función Principal.

7.1.8.1. Read()

El objetivo de esta función es leer el voltaje entre el pin Control Pilot y la tierra, que es donde se realiza la comunicación entre el Arduino y el vehículo, en base al protocolo
IEC 61851-1

Para eso, se realizarán sucesivas medidas hasta que el voltaje leído esté dentro de un entorno esperable. Y esas medidas, tendrán que realizarse dentro de un intervalo máximo de tiempo determinado por la constante **DelayRead**.

Se guarda en la variable **AnalogVout** el valor de voltaje medido por el pin analógico $A3$. Como el pin analógico, mapea los valores entre 0 V y 5 V a valores entre 0 y 1023, entonces para llevar el dato a un valor de voltaje en el rango entre 0 V y 5 V, que es la que se desea utilizar, el dato medido tiene estar que multiplicado por 5 y dividido por 1023

Por lo tanto, la formula para la medida será **Vout = AnalogVout*(5/1023)*3 + 0.7**.

Vale aclarar, que como se desea medir voltajes de hasta 12 V, en la plaqueta donde se implemento el circuito SAE, también se colocó un divisor de tensión, para que la tensión en el punto de medida este en el rango de 0V a 5 V. Por eso, se multiplica el valor medido, por un factor de 3, porque ese fue el divisor de tensión que se implemento.

Mientras que, se suma 0,7 V porque hay una caída de voltaje de un diodo, en el punto donde el Arduino mide el voltaje.

Una vez medido el voltaje, cambiada la escala y guardado el dato en la variable **Vout**, se verifica si ese valor está dentro del entorno de 12 V o del entorno de 9 V o del entorno de 6 V. Mientras no esté dentro de ninguno de esos entornos, se vuelve a tomar un nuevo dato del pin analógico.

Y cuando un dato de **Vout** este dentro de alguno de esos entornos, la función Read deja de ejecutarse. Dejando la variable **Vout** con el dato actualizado, para que la utilicen otras funciones dentro del archivo Save.cpp.

Como la función *Read()* es ejecutada en el *Loop()* principal, el dato **Vout** es actualizado constantemente.

7.1.8.2. DutyCicle().

Esta función devuelve como resultado un entero: el valor del **DutyCicle** de la señal PWM adecuada para el cable del conector del vehículo que se acaba de conectar, en base al protocolo IEC 61851-1 Anexo B.

Para eso, se realiza una medida del voltaje entre la resistencia del cable del conector -que está estandarizada, para indicar cuánta corriente admite el cable- y una resistencia conocida, colocada en el circuito diseñado.

Se guarda en la variable **AnalogWire_value** el valor de voltaje medido por el pin analógico $A2$. Y como el pin analógico, mapea los valores entre 0 V y 5 V a valores entre 0 y 1023, entonces para llevar el dato a la escala entre 0V y 5V, que es la que se desea utilizar, el dato medido tiene que estar multiplicado por 5 y dividido por 1023.

Por lo tanto, el resultado de la medición esta dado por la relación **Wire_value = (Analog-Wire_value)*(5/1023)**.

Con el voltaje medido **Wire_value**, se sabe cuál es la resistencia estandarizada que tiene el cable. Por lo que el siguiente paso, es verificar si ese valor de resistencia, corresponde a una corriente máxima admisible de 13 A o de 20 A o de 32 A o de 63 A.

En el protocolo IEC 61851-1 Anexo B (al igual que en el SAE J1772) se establece una relación biunívoca entre la máxima corriente admisible y el **DutyCicle** que corresponde a dicha corriente. La relación establece que el **DutyCicle** valdrá:
$(I_{MAX}/0{,}6)$, si la corriente es entre 6 A y 51 A;
mientras que $(I_{MAX}/0{,}6) + 64$, si es una corriente entre 51 A y 80 A.

Como el **DutyCicle** es la relación de tiempo en que la señal PWM está en ON con respecto al período de la señal; entonces el parámetro **DutyCicle** es porcentual. Para que deje de ser un valor porcentual, hay que dividirlo entre 100.

Para poder generar una señal digital en el **Pin_PWM** del Arduino, toda señal tiene que estar en una escala entre 0 y 255. Por lo que el parámetro **DutyCicle** hay que cambiarlo de escala, multiplicandolo por 255.

Finalmente, por la forma que fue diseñado el circuito SAE, el PWM a la entrada (generado por el Arduino) será el inverso al PWM a la salida del circuito SAE. Entonces, para obtener el **DutyCicle** deseado a la salida del circuito SAE, el Arduino tendra que generar una señal PWM con el **DutyCicle** inverso.

La función *DutyCicle()*, retornará el **DutyCicle** de la señal PWM que el Arduino generará:

- Si I_{MAX} del cable está en el entorno de 13 A, retornará: **255*(1 - (13/0.6)*(1/100))**
- Si I_{MAX} del cable está en el entorno de 20 A, retornará: **255*(1 - (20/0.6)*(1/100))**
- Si I_{MAX} del cable está en el entorno de 32 A, retornará: **255*(1 - (32/0.6)*(1/100))**
- Si I_{MAX} del cable está en el entorno de 63 A, retornará lo mismo que con 32 A. Porque aunque se sepa que el cable soporta hasta 63 A, el SAVE tiene que indicarle al vehículo, que la máxima corriente a entregar podrá ser de 32 A.
- Si I_{MAX} del cable está fuera de esos entornos, entonces la función *DutyCicle()* devolverá un **CodigoError**.

7.1.8.3. ReadCard().

Esta función devuelve como resultado un booleano: **True** o **False** a la pregunta de si se leyó una tarjeta. En el caso de que lea una tarjeta, se envía a la Raspbery el código de la misma, mediante la comunicación serial ya establecida. Para verificar la presencia de una tarjeta y para leer el código de la misma, se hizo uso de la librería <**MFRC522.h**> con la que se define un objeto **mfrc522** con el cual que se puede acceder a dichas dos funciones.

7.1.8.4. StatusDisplay().

Esta función chequea el valor de la variable **DispStateValue**, y dependiendo de su valor, muestra en el Display cierto mensaje, por un intervalo de tiempo igual a **Delay**.

Para mostrar cada mensaje en el Display, se utiliza la función ***message(Firstmessage, Secondmessage)***.

7.1.8.5. message(Firstmessage, Secondmessage).

Esta función recibe como argumento, dos *String*, que serán los mensajes en el Display.

El primer paso es borrar el mensaje anterior. Luego se setea la posición del primer mensaje, y se imprime en la pantalla. Después se repite el procedimiento para el segundo mensaje. Quedando en la pantalla, el primer mensaje impreso en la mitad superior de la pantalla y el segundo, en la parte inferior.

Para borrar, setear posicion e imprimir el mensaje, se hizo uso de la librería <**LiquidCrystal_I2C.h**> con la que se define un objeto **lcd** con el que se puede acceder a las tres funciones mencionadas.

7.1.8.6. Init()

Esta función se ejecuta una sola vez, en el Setup() del codigo principal. Se utiliza para inicializar algunas funciones y configura los pines necesarios para el resto del código.

Primero se inicializa la comunicación serial, seteando la velocidad en baudios de la transmisión de datos, que en este caso será entre la Raspberry y el Arduino. Para esta funcionalidad, no fue necesario incluir una librería.

Luego se inicializa la conexión con el modulo RFID (lector de tarjeta) y el Arduino. inicializando el bus SPI y el RC522 (este ultimo mediante el objeto mfrc522). Para esta funcionalidad, fue necesario incluir las librerías <**SPI.h**> y <**MFRC522.h**>.

A continuación se inicializa la conexión con el Display, mediante el objeto lcd, que para poder utilizarlo fue necesario incluir la librería <**LiquidCrystal_I2C.h**>.

Después se configuran los dos pines de salida, uno para generar la señal PWM y otro para comandar el relé. Y también los dos pines analógicos de entrada, para leer el voltaje del CP para la comunicación con el vehículo y para leer el voltaje del PP para establecer el **DutyCicle** correspondiente a la corriente máxima que admite el cable a utilizar.

Una vez configurados los pines, se setea el valor del pin que comanda el relé en HIGH, para asegurarse de que el relé esté abierto.

Mientras que las últimas dos lineas, son para inicializar una funcionalidad de la librería <**PWM.h**> que permite configurar la frecuencia del pin que genera el PWM, para asegurarse que dicho PWM sea una señal de 1kHz. Para así, poder cumplir con el protocolo el IEC 61851-1 (y también con el protocolo SAE J1772).

CAPÍTULO 8

DISEÑO FINAL DEL SAVE.

En este capítulo se analizarán todos los componentes utilizados para el diseño del SAVE. Para ello se dividirá el SAVE en tres partes según sus funciones: Primero se verá el circuito de potencia, una segunda parte correspondiente al circuito de control y una tercera parte es la encargada de alimentar otros componentes como las dos fuentes de 12V, la Raspberry, el Arduino y otros dispositivos.
En la figura 8.1, se presentan los componentes del SAVE desarrollado:
En el **recuadro rojo** está la parte encargada de realizar las medidas de de las magnitudes eléctricas de interés, energía tensión, intensidad de corriente, etc. En el **recuadro amarillo** se tiene los componentes de sistema trifásico de potencia y la alimentación monofásica a partir de la cual se alimenta el circuito de control. Finalmente en el **recuadro azul** se tiene el circuito de control responsable de la comunicación con el vehículo y con el centro de control de carga. En a tapa se puede apreciar la pantalla LCD, el lector de tarjeta RFIT y el conector tipo 2 hembra.

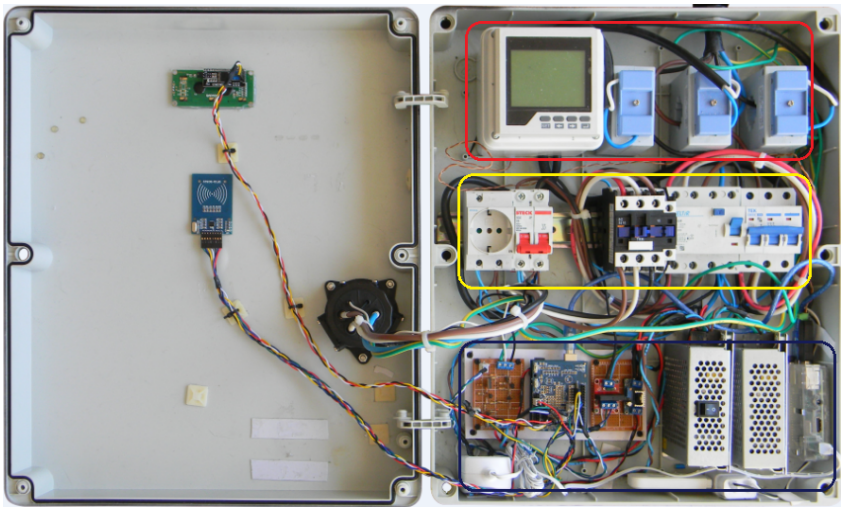


Figura 8.1: SAVE Diseñado

8.1. Circuito de Potencia.

Se parte de la base que la alimentación del SAVE es una linea trifásica con neutro, por lo tanto, los componentes que aparecen en este circuito son todos trifásicos. A continuación se listan los componentes utilizados.

- Llave termo-magnética.
- Llave diferencial.
- Medidor de energía.
- Convertidor de corriente.
- Contactor.
- Conector (hembra) tipo 2.

En la siguiente figura se puede ver el unifilar de este circuito:

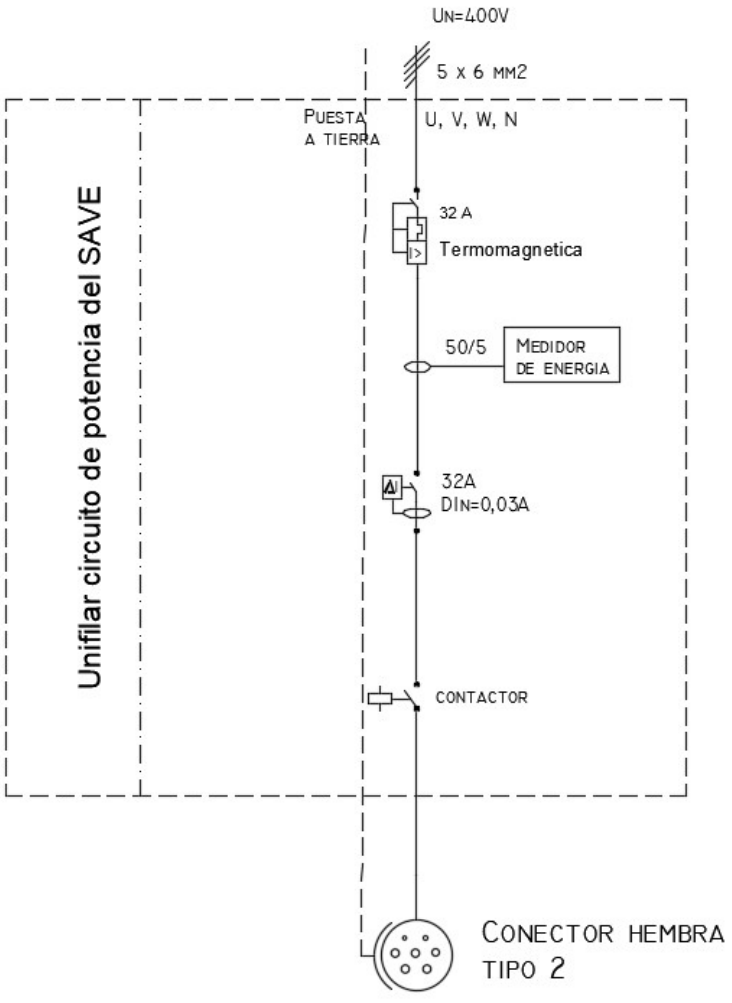


Figura 8.2: Unifilar circuito de potencia.

8.2. Circuito de Control.

El circuito de control es un relé de estado sólido, que comanda al contactor, energizando o desenergizando su bobinado primario.

El relé está alimentado por 5 V y comandado por un pin digital del Arduino. Mientras el pin del Arduino este HIGH, el relé permanece abierto, porque no habrá diferencia de voltaje entre la alimentación y el pin de comando, entonces el relé queda desenergizado y la bobina del contactor también. Por lo que, en dicho caso, el circuito de potencia permanece abierto.

Mientras que cuando el pin digital en LOW, el relé cerrará porque habrá diferencia de voltaje entre la alimentación y el pin de comando. Y en dicho caso, la bobina del contactor se energizará y cerrara el circuito de potencia.

El beneficio de esta lógica, es que si el Arduino pierde su alimentación, entonces el relé dejará de estar alimentado y permanecerá abierto. Por lo que el contactor va a desenergizarse y abriría el circuito de potencia.

La figura 8.3 muestra el circuito descripto anteriormente.

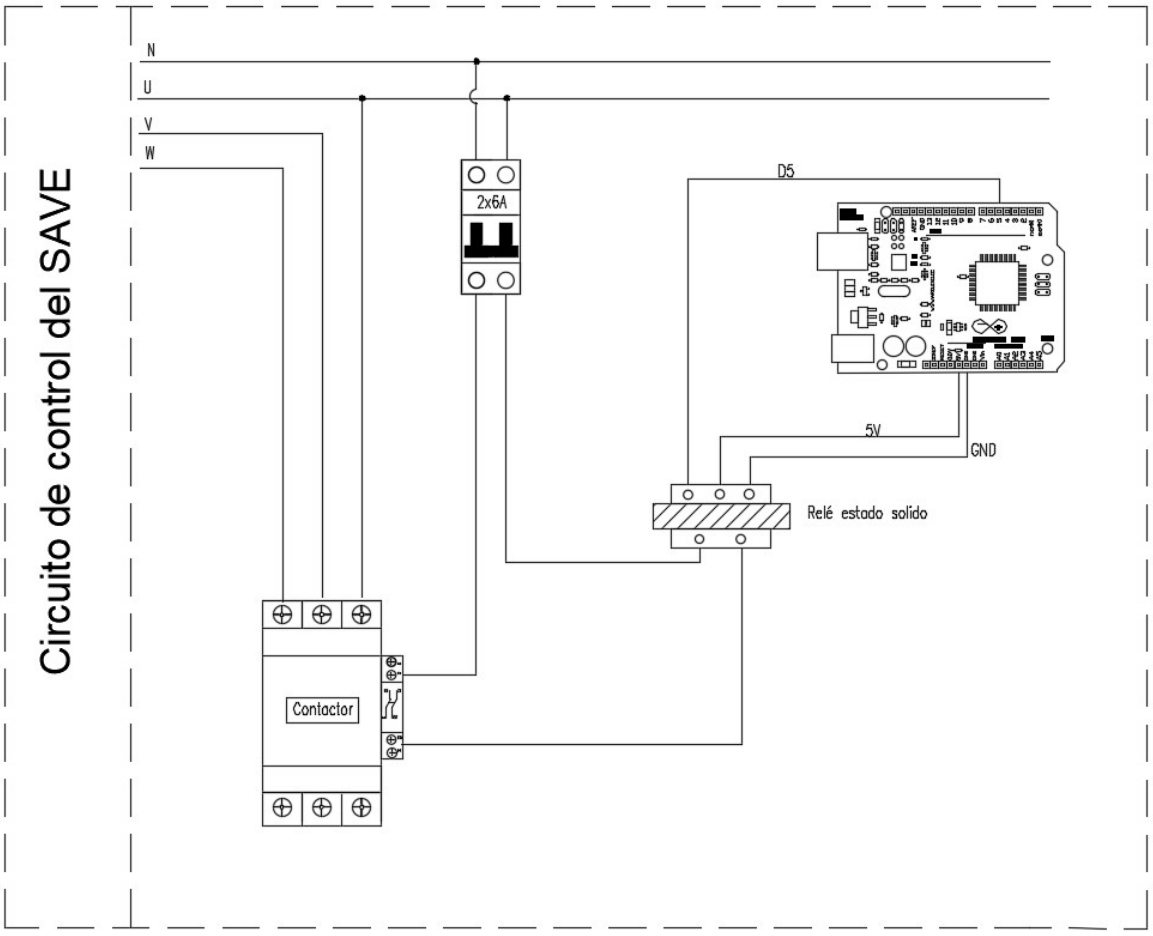


Figura 8.3: Unifilar circuito de control.

8.3. Circuito de Alimentación.

Esta parte del circuito describe la alimentación de los componentes. En primer lugar aparece la alimentación de la Raspberry la cual se realiza a través de un convertidor de 230 $VAC/5\,VDC$. Luego se tiene la alimentación de las dos fuentes de 12 V las cuales alimentan por un lado al circuito IEC y por otro al Arduino, pasando antes por un regulador de tensión, para llevar la misma a 5 V. Luego a través de un convertidor de tensión de 230 $VAC/5\,VDC$ se alimenta el módem WiFi, siendo imprescindible para la conexión a la red VPN. Por último aparece un toma corriente monofásico el cual fue incluido dentro del SAVE para contar con un toma accesible al momento de realizar ensayos, reparaciones, etc.

A su vez se muestra como es la conexión entre el Arduino y Raspberry (por intermedio de la conexión USB) y de la comunicación entre la Raspberry y el medidor el cual se conecta a través de un conversor RS-485 conectado a uno de los puertos USB de la Raspberry.

La figura 8.4 muestra el circuito descripto anteriormente.

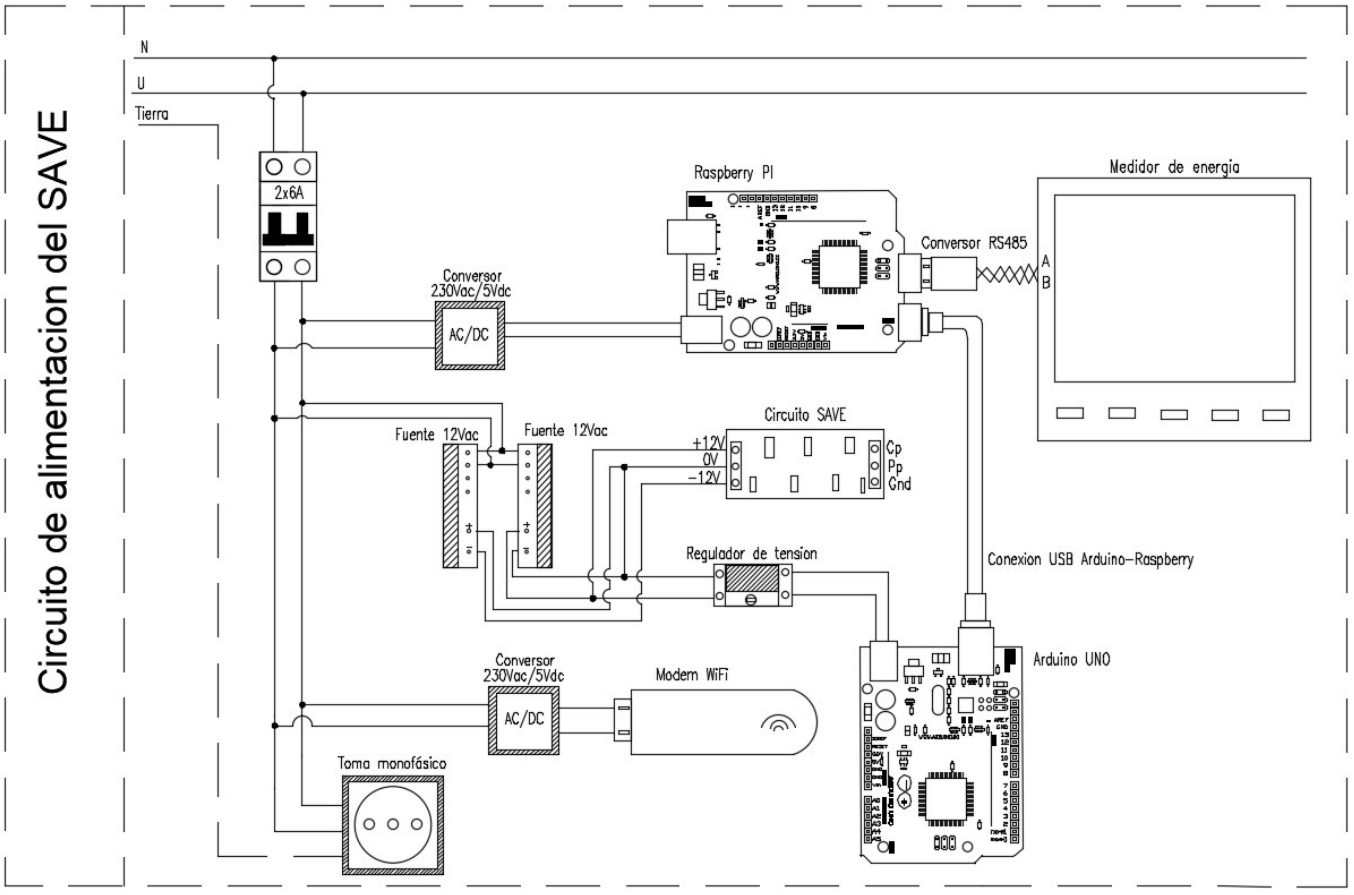


Figura 8.4: Unifilar circuito de alimentación.

8.4. Circuito de Conexión de Componentes del Arduino.

En este circuito se ilustra como es la conexión entre el Arduino y los componentes que interactúan con él. Cabe resaltar que para facilitar las conexiones se utilizo un shield para Arduino el cual se monta sobre el mismo habilitando mayor numero de pines y una mejor distribución de ellos. La figura 8.5 muestra como es esta conexión.

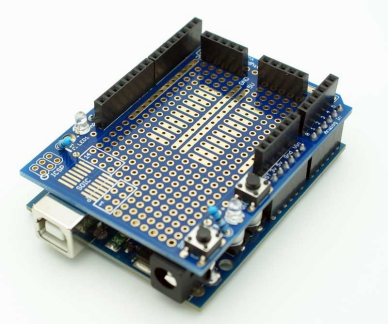


Figura 8.5: Arduino + shield.

En primer lugar se tiene la pantalla LCD 16x2 para la cual se utilizó un módulo I2C para su conexión. La tabla 8.1 describe la conexión entre pines.

Pin Arduino	Pin Módulo I2C
Gnd	Gnd
5V	Vcc
A4	SDA
A5	SCL

Tabla 8.1: Conexión entre pines Arduino - I2C.

También se tiene la conexión con el lector de tarjeta RFID-RC522 que se realiza según la tabla 9.1.

Pin Arduino	Pin Módulo RFID
10	Sda
13	Sck
11	Mosi
12	Miso
-	Rq
Gnd	Gnd
9	Rst
3.3V	3.3V

Tabla 8.2: Conexión entre pines Arduino - RFID.

El hilo de control CP se conecta al pin A3 del Arduino, mientras que el hilo de proximidad PP se conecta al pin A2 del Arduino. En el circuito de la figura 6.1 se puede ver la ubicación de la conexión CP y PP dentro del circuito IEC.

El pin digital 3 del Arduino es el encargado de generar la señal PWM. En el circuito de la figura 6.1 se puede ver la ubicación de la conexión Vin dentro del circuito IEC.

El pin digital 5 del Arduino es el encargado de comandar el relé de estado solido que a su vez comanda el contactor. En la figura 8.3 se puede ver la conexión del pin D5 dentro del circuito de control.

Todas las conexiones descriptas anteriormente se pueden ver en la figura 8.6.

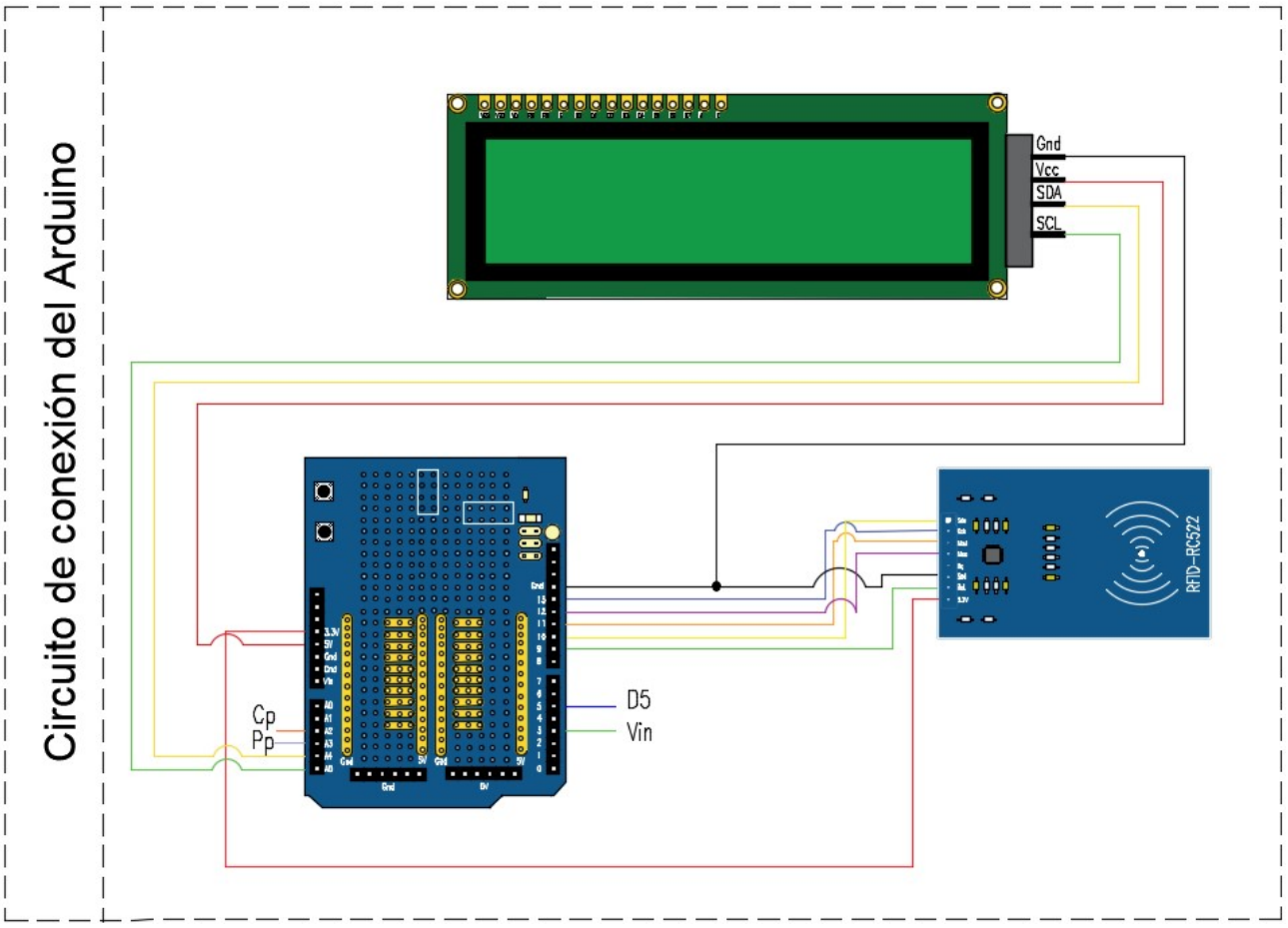


Figura 8.6: Conexión de Arduino y sus componentes.

CAPÍTULO 9

PRUEBAS FINALES E INSTRUCTIVO DE USO.

9.1. Pruebas Finales.

Una vez construido el SAVE se llevaron a cabo una numerosa cantidad de pruebas para chequear el buen funcionamiento y la fiabilidad del mismo.

Las primeras pruebas fueron estrictamente sobre el comportamiento del software implementado, tanto para chequear que cumpla de forma fidedigna la comunicación SAVE-servidor - según establece el protocolo OCPP 1.6 - así como para testear la capacidad de respuesta ante situaciones anormales.

Algunas de las pruebas realizadas, con dicho objetivo, fueron:

- Perdida del mensaje de respuesta del servidor.

  En este caso, la conexión con el servidor se mantiene abierta correctamente, pero el problema que ocurre es que cuando se envía un mensaje, la respuesta del servidor nunca llega. Lo que se planteó como solución fue que se espera un tiempo (se probó con 10 segundos, pero es configurable) y si la respuesta no llega, se vuelve a enviar el mismo mensaje.

  Esto se repite una cantidad acotada de veces (se probó con 3) y si la respuesta nunca llega, el SAVE pasa al estado de fuera de servicio.

- Chequeo del comportamiento del SAVE a la hora de perder la conexión con el servidor.

  Para estas pruebas se mantenía el SAVE funcionando en condiciones normales y, en determinado momento, se apagaba la señal WiFi provocando que se pierda la comunicación con el servidor. Esta prueba se realizó en todas las instancias de un proceso de carga, es decir, antes de que llegue un usuario, con el vehículo cargando, con un vehículo conectado esperando para cargar, etc.

  Para todos los casos se comprobó que la respuestas fue la deseada, por lo tanto, si la comunicación con el servidor se retomaba en menos un tiempo máximo (se probó con 15 segundos, pero es configurable), entonces el programa continuaba corriendo de forma normal y en el estado donde se había perdido la comunicación; sin afectar en ningún aspecto el proceso de carga del VE.

En el caso que la conexión con el servidor no se retome en ese tiempo, el SAVE corta inmediatamente la carga (si es que estaba cargando el VE) y pasa al estado fuera de servicio, indicando dicho estado en el display. Además, al momento de retomar la conexión con el servidor, se le envía un mensaje de **StopTransaction** para cerrar la transacción que haya quedado trunca (en caso que la perdida de servidor haya dejado una sesión de carga sin terminar).

Luego se realizaron pruebas de hardware, para verificar de forma fidedigna la comunicación SAVE-Vehículo - según establece el protocolo IEC 61851-1 en Anexo A - así como para testear la capacidad de respuesta ante situaciones anormales.

- Se comenzó realizando ensayos con las resistencias estandarizadas en el protocolo, para simular la conexión con un vehículo. De esa manera se pudo medir la calidad de la señal PWM con osciloscopio y controlar el momento de generar dicha señal para iniciar la comunicación. Mientras que el circuito de potencia, durante estos ensayos era alimentado por un sistema monofásico, y se ensayaba con cargas monofásicas; para observar efectos de la conexión y desconexión del contactor mientras hay consumo de corriente.

  Una vez cumplido el protocolo, se realizaron pruebas para estresar al circuito: conectando y desconectando la carga intempestivamente.

  Es decir, se iniciaba una sesión hasta el estado donde el contactor estuviese cerrado; y se conectaba o desconectaba una carga comprobándose que el SAVE se mantenía estable y funcionando de forma normal.

  También se realizó la desconexión del conector Tipo-2 del vehículo, de forma intempestiva, mientras transcurría una carga. Y se comprobó que el SAVE actuó de forma esperada, cortando inmediatamente la energía y enviando al servidor un mensaje de **StopTransaction** para finalizar la sesión de carga.

- Como prueba final se llevaron a cabo tres ensayos del SAVE con un vehículo eléctrico real, proporcionado por UTE. Los primeros dos ensayos, se realizaron con un vehículo que consumía como máximo 16 A y el último consumía como máximo 32 A, ambos consumos monofásicos.

  En el **primer ensayo** se constató que el vehículo detectaba la señal PWM del SAVE, pero que un instante después, se mantenía a 9 V, en un estado que no posibilita el inicio de la carga. Luego se revisaron detalles alimentación del circuito de electrónica porque, cuando se abría el contactor, se reseteaba la Raspberry.

  A pesar de ello, el PWM implementado en este proyecto tenía tiempo de subida y bajada que cumplían con todas las especificaciones de la norma. Luego se hicieron las mismas pruebas con un SAVE fabricado por Circontol, este tenia el doble de tiempo en la señal de subida y bajada del PWM estando fuera de la norma. Sin embargo, el vehículo se comunicaba a través del PWM con el SAVE sin problemas.

  Luego de ese ensayo, se pudo corregir el cálculo del **DutyCycle** de la señal PWM implementada. El error fue no considerar que el protocolo daba el dato de **DutyCycle**, como porcentual. A su vez, el reseteo de la Raspberry se pudo corregir, alimentando la misma de forma independiente mediante un conversor de energía. Además el contactor, se empezó

a comandar mediante un relé de estado sólido, para evitar el efecto del pico de voltaje que se observó en la bobina de otro modelo de relé, cuando se desenergizaba.

En el **segundo ensayo** se constató que el VE reconoció perfectamente la señal PWM enviada por el SAVE y se pudieron realizar todos los procesos de carga, pasando por cada una de las etapas, sin dificultades. También se chequeo (utilizando una pinza amperimétrica) que el valor de corriente que tomaba el vehículo correspondía al PWM que el SAVE enviaba como corriente máxima a consumir.

Igualmente, el medidor de energía obtenía alrededor de la mitad del valor de corriente que efectivamente estaba consumiendo el vehículo. Debido a que no se contaba con el manual del medidor de energía en ese momento, recién se pudo verificar la configuración unos días después.

En la figura 9.3 se pueden ver algunas fotografías del momento en que se realizaron las pruebas con el SAVE y un VE.

(a) VE conectado (b) Consumo (c) SAVE

Figura 9.1: Carga de un VE.

- Al obtener el manual, se pudo comprobar que la relación de vueltas de los transformadores de corriente estaba bien ingresada en el medidor de energía, porque los transformadores indicaban 50/5 y en el medidor se ingresó una relación de 10. Por otro lado, el manual indicaba que existían dos formas de realizar el promedio trifásico de la corriente, y por lo tanto se configuró la otra forma de realizar el promedio, con la expectativa de que eso resolviera el problema.

- En el **último ensayo** con un vehículo real, que consumía como máximo 32 A, se pudo comprobar que las diferentes variantes del proceso de carga se realizaban sin problemas. Pero se mantuvo el error en el valor que obtenía el medidor de energía, por lo se ensayaron los transformadores de corriente (de origen chino), porque la etiqueta podría tener la relación equivocada. Gracias a la sugerencia de Ing. Pablo Toscano, se pudieron ensayar los transformadores de corriente, obteniendo los siguientes resultados:

Corriente en el primario	Corriente medida en el secundario	Corriente real en el secundario	Relación de transformación
8,0	2,1	0,42	19
15,0	3,9	0,78	19
19,5	5,2	1,04	18,75
25,2	6,7	1,34	18,75
30,0	8,0	1,6	18,75
35,5	9,5	1,9	18,68
38,2	10,2	2,04	18,72
44,1	11,8	2,36	18,68

Tabla 9.1: Relación de transformación de los transformadores de corriente

Como se puede observar, los transformadores tenían el etiquetado con un valor equivocado y fuera del margen de error esperable. Por lo tanto, alcanzó con setear el medidor de energía con una relación de transformación de 19 y los valores de corriente obtenidos resultaron aceptables.

(a) SAVE diseñado (b) Conexionado

Figura 9.2: Ensayo de los transformadores de corriente.

(a) VE de UTE (b) Conexión al SAVE

Figura 9.3: Último ensayo con Vehículo Eléctrico.

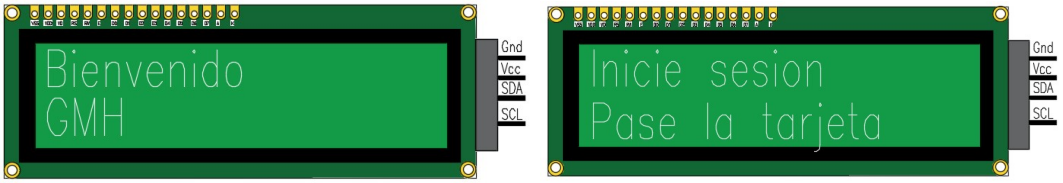
9.2. Instructivo de Uso.

Es esta sección se hará una explicación sobre el uso del SAVE por parte de un usuario. Para facilitar su uso, el SAVE cuenta con un display donde a través de mensajes va interactuando con el usuario guiándolo en el proceso de carga e informando las instancias de la misma.

Al momento de ser energizado el SAVE entra en un estado "Fuera de servicio" que se mantiene mientras sus componentes (Arduino, Raspberry) se inician y comienzan a correr el software correspondiente.

Figura 9.4: Pantalla: fuera de servicio.

Una vez que todos los componentes se iniciaron correctamente y se logró la comunicación con el servidor el SAVE está listo para recibir a un usuario. Este estado el SAVE lo manifiesta escribiendo en su pantalla el mensaje "Bienvenido GMH, Inicie sesión pase la tarjeta".



(a) Bienvenida. (b) Inicie sesión.

Figura 9.5: Pantallas de bienvenida.

Como se puede ver en la figura 9.5 lo primero que ve un usuario al momento de enfrentarse al SAVE es el mensaje de bienvenida, seguido de la instrucción de pasar la tarjeta para comenzar una carga.

Por lo tanto, para continuar con el proceso de carga el usuario debe pasar una tarjeta por el lector de tarjetas. Y la ubicación del lector de tarjeta está indicada con una imagen en el exterior del SAVE que hace alusión al lector.

Figura 9.6: Señalización del lector.

Una vez que es pasada una tarjeta el SAVE muestra en su pantalla el mensaje “tarjeta leída”. Esto se puede ver en la siguiente figura.

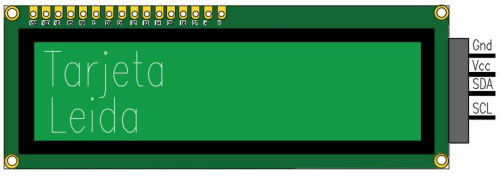


Figura 9.7: Pantalla: tarjeta leída.

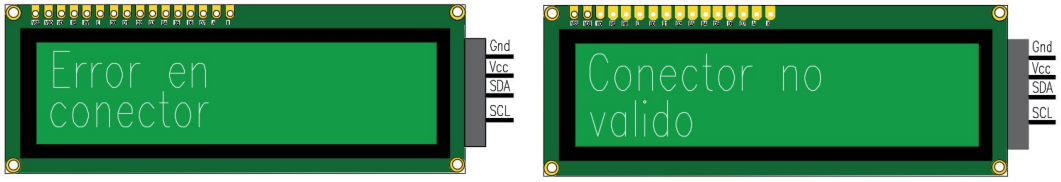
En el caso que la tarjeta no sea aceptada se mostrara en la pantalla el mensaje “Tarjeta rechazada, pase otra tarjeta” y se retornara al mensaje de bienvenida.

Figura 9.8: Pantalla: tarjeta rechazada.

En el caso que la tarjeta sea válida, el mensaje que se despliega en el display es “conectar el vehículo”.

Figura 9.9: Pantalla: conectar vehículo.

Si el conector que se utiliza no es válido para la carga se muestra en el display el mensaje "error en conector, conector no valido". Este mensaje se desplegará hasta que el conector es desconectado.



(a) error. (b) conector.

Figura 9.10: Conector no valido.

Si el conector que se coloca es válido, el mensaje que se muestra en la pantalla es "procesando la solicitud, aguarde".

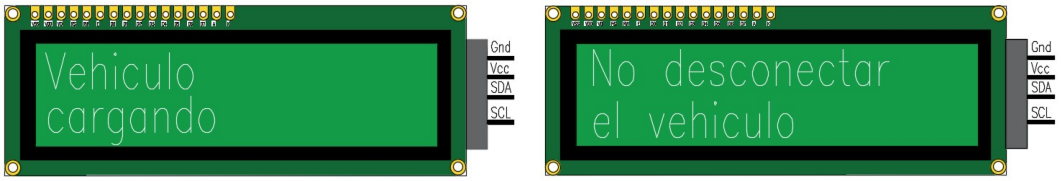
Figura 9.11: Pantalla: de espera.

En esta instancia el SAVE pide autorización al servidor de UTE para comenzar una sesión de carga, en caso que no es autorizada se despliega en la pantalla el mensaje "carga rechazada, desconectar el vehículo".

Figura 9.12: Pantalla: sesión rechazada

En el caso que el servidor acepte la solicitud de inicio de sesión de carga la carga comienza, desplegándose en la pantalla el mensaje "vehículo cargando, no desconectar el vehículo, pase la

tarjeta para detener". Hasta que no se pase la tarjeta este mensaje se mantiene en pantalla.



(a) cargando. (b) no desconectar.

Figura 9.13: Pantallas en carga.

Figura 9.14: Pantalla: mensaje para detener.

Una vez que se pasa la tarjeta por el lector para detener la sesión de carga se despliega el mensaje "tarjeta leída" y si la tarjeta es la misma que inicio la sesión de carga se detiene la sesión y se despliega en la pantalla el mensaje "carga finalizada, desconectar el vehículo".

Figura 9.15: Pantalla: carga finalizada.

Una vez que se desconecta el vehículo, el SAVE a la espera de un nuevo usuario vuelve a desplegar en la pantalla el mensaje de bienvenida de la figura 9.5.

Cabe resaltar que si en cualquier momento del proceso sucede algo inesperado (se pierde la conexión con el servidor, falla la comunicación entre la Raspberry y el Arduino, etc.) el SAVE entra a un estado de fuera de servicio desplegando en la pantalla el mensaje de "fuera de servicio" de la figura 9.4.

CAPÍTULO 10

ESTUDIO TEÓRICO DEL CARGADOR BIDIRECCIONAL.

En el presente capítulo se estudian los requerimientos necesarios para que el SAVE desarrollado pueda implementar la bidireccionalidad. Es decir, que el SAVE permita el flujo de potencia activa en ambos sentidos: hacia y desde la batería del vehículo.

10.1. Motivo del Estudio.

A partir de conclusiones de diversos científicos, de que está ocurriendo un cambio climático real y que la causa principal del calentamiento global es la contaminación producidas por las actividades humanas [IPCC 2001] (ver [12],[13]), se lograron acuerdos internacionales como el COP-21 de París (ver [14]), en el cual cada nación se auto-compromete a reducir emisiones de dióxido de carbono (CO2); sea en el transporte, la industria o el hogar.

En cuanto al transporte, hay países en los cuales se propusieron metas para vender solamente vehículos eléctrico o híbridos. En el caso de Reino Unido y de Francia, se propuso prohibir la venta de vehículos de combustión para el 2040 (ver [15]). Por lo que la venta de vehículos eléctricos o híbridos ya hace unos años viene en aumento.

Por otro lado, en la última década se observó un aumento en la instalación de diversas formas de generación eléctrica, a partir de fuentes renovables no-poluentes.

El problema de generación de energía eléctrica, a partir de fuentes renovables, es que esta energía es "no-despachable" (ver [16]). Es decir, mientras están disponibles es posible que nadie las requiera. Y que cuando son requeridas, es posible que no estén disponibles.

Por ejemplo, si el viento sopla más fuerte durante la madrugada, es posible que se pueda tener mayor generación de energía en ese horario. Pero en la madrugada, es muy baja la demanda y por eso en ese horario no se puede despachar todo lo que se podría generar. Por lo tanto, la oferta de energía eléctrica a partir de fuentes renovables está limitada por ser "no-despachable".

Mientras que la demanda, depende del conjunto de individuos que requieren electricidad en cada instante. Si los clientes pagan por dicho servicio, la empresa de energía eléctrica está comprometida a suministrarla en cualquier momento, con la mayor calidad y confiabilidad posible.

Es por esto, mientras se invierte en cambiar la matriz energética a fuentes renovables no-poluentes, al mismo tiempo se buscan soluciones para la gestionar eficientemente la red eléctrica,

con el objetivo de que la demanda esté siempre satisfecha, que se reduzcan las emisiones de CO2 pero evitando que se desperdicie la energía eléctrica generada.

Un caso emblemático de dicha problemática es el de Alemania (ver [16]) que, desde 2010 hasta 2016 multiplicaron la cantidad de parques eólicos. Y en ese país, mayo de 2016, en ciertos momentos, el 90 % del suministro de energía eléctrica provino de fuentes renovables. Sin embargo, las emisiones de carbono aumentaron ya que se recurrió a combustibles fósiles para satisfacer la demanda en todo momento.

Es decir, se desecha gran parte de la energía generada por fuentes renovables porque no coincide con el momento en que hay demanda. Y cuando hay demanda, por no haber suficiente oferta, se quema carbón para generar energía eléctrica.

Una manera de solucionar ese problema es utilizar bancos de baterías, para almacenar la energía eléctrica proveniente de fuentes renovables, que en su momento no fue consumida. Y luego despachar dicha energía almacenada, en el instante que la demanda lo requiera. Esas baterías pueden ser instaladas en edificios, pero suponen un gran costo.

Con esos mismos fines, **se pueden utilizar las baterías de los vehículos eléctricos**, si es que el vehículo permite la descarga de su batería para suministrar energía eléctrica a la red o a un hogar.

A diferencia de consumidores usuales, el usuario del vehículo eléctrico quiere que su auto este cargado al momento de comenzar a conducir, pero mientras tanto, es posible cargar o pausar la carga, lo que es denominado la "carga dinámica". Si los vehículos eléctricos son una cantidad significativa de la demanda total de la red eléctrica, este tipo de "carga dinámica", podría servir para gestionar la curva de demanda de la red.

Al utilizar las baterías de VE como bancos de baterías **gestionable**, la empresa prestadora del servicio eléctrico consigue: una mayor eficiencia, menor pico de demanda, menor contaminación, e inclusive menores costos. Mientras que el cliente puede recibir descuentos en la tarifa por los servicios prestados, y si el precio de la energía eléctrica fuese variable en el tiempo, el cliente podría cargar su VE a bajo precio y descargar su VE cuando el precio está al alza.

Por lo mencionado anteriormente, cobra relevancia la posibilidad de **descargar la batería del vehículo eléctrico, como método de gestión inteligente de la curva de demanda**.

10.2. Introducción.

Durante la **carga** de la batería de un vehículo eléctrico (VE), el flujo de potencia activa va desde la red eléctrica hacia la batería. Mientras que durante la **descarga**, el flujo de potencia activa va desde la batería hacia la red eléctrica (ver [18]).

La carga de un vehículo eléctrico, puede ser gestionada inteligentemente, mediante lo que se denomina carga dinámica. Que implica que haya una comunicación entre el cargador (SAVE) y una central de gestión, para que la central pueda indicarle al SAVE si esta habilitado a cargar en cierto horario, y que la central pueda variar la corriente máxima que dicho SAVE le pueda entregar al vehículo en dicho horario.

Esto ya puede ser implementado con un SAVE unidireccional, como el diseñado en este proyecto, pero agregando la posibilidad de recibir mensajes desde el centro de control de carga, que cambien la configuración del SAVE para que entregue una corriente máxima distinta.

Otra forma de gestión inteligente de la curva de demanda, es la descarga de baterías de de vehículos eléctricos. En la biblografía citada (ver [19] y [22]), se menciona que este tipo de descargas, puede utilizarse para diversas funcionalidades que se categorizan según el tipo de carga que lo consuma:

- Vehicle-to-Grid (V2G): si se suministra energía eléctrica desde la batería del VE a la red.
- Vehicle-to-Home (V2H): si se suministra energía eléctrica desde la batería del VE al hogar.
- Vehicle-to-Live (V2L): si se suministra energía eléctrica desde la batería del VE a los dispositivos eléctricos directamente, funcionando como fuente de energía de emergencia.

Para referirse genéricamente a cualquiera de dichos métodos, se utiliza V2X (Vehicle-to-X).

Las tecnologías V2H y V2G han comenzado a utilizarse de manera práctica, con el objetivo de controlar la reducción de picos de demanda, la distribución de energía en caso de falla eléctrica y la estabilización de la energía de la red. Se han llevado a cabo varios experimentos de demostración y tales tecnologías ahora se usan en parte (ver [19]).

Sabiendo que la tendencia para conseguir un control inteligente del consumo-demanda de energía eléctrica, tanto en V2H y V2G, es esperable que se utilice como intermediario un SAVE.

10.3. ¿Qué es un SAVE Bidireccional?.

A un cargador (SAVE) se le denomina unidireccional cuando permite solamente la carga de la batería del VE y es bidireccional cuando permite la carga y la descarga.

El SAVE unidireccional, permite cargar la batería en alguna de las siguientes formas:

- 1) Suministrando corriente en DC al VE -a partir de la alterna de la red, la convierte con un AC/DC a corriente directa- y en ese caso, cargaría la batería directamente. Igual, siempre se utiliza un controlador del estado de carga (SoC).
- 2) Suministrando corriente alterna AC al VE. Y dejando la conversión para el vehículo, que para este caso, requiere tener integrado un conversor AC/DC.

Se le denomina SAVE bidireccional, cuando:

- 3) Cumple la condición (1) y además, puede consumir del VE corriente en DC y mediante un inversor suministra la corriente alterna a la red.
- 4) Cumple la condición (2) y además, puede consumir del vehículo eléctrico la corriente alterna AC. En este caso el VE debe tener integrado un conversor DC/AC bidireccional.

10.4. Diferencias entre un SAVE en DC o en AC (con conversor unidireccional o bidireccional)

Para las cargas en corriente directa (DC), se precisa rectificar la corriente alterna, proveniente de la red, con un conversor AC/DC que está fuera ("off-board") del vehículo eléctrico y dentro del SAVE. Mientras que para las cargas en AC, el conversor está fuera del SAVE y está a bordo del vehículo -los denominados "On-Board Chargers"(OBC)- que consiste en una etapa AC/DC que rectifica la corriente y con un DC/DC controla el nivel de voltaje. Entonces es notorio que en cargas en AC, el diseño del SAVE es más sencillo, porque no tiene un conversor integrado ni tiene que comandar el OBC.

En cuanto a diferencia entre un SAVE unidireccional o bidireccional (sea en DC o en AC), depende de que el conversor permita el flujo unidireccional o bidireccional de potencia activa.

Para habilitar la bidireccionalidad en DC, un SAVE unidireccional en DC, tendría que cambiar el diseño de sus conversores y aplicar un control más complejo[1].

Por otro lado, para habilitar la bidireccionalidad en AC, los fabricantes de dichos vehículos con OBC tienen que implementar de un conversor bidireccional (y el SAVE solo tendría que comunicarse con el VE de forma controlada para aplicar la bidireccionalidad). Actualmente, existen OBC que utilizan llaves de potencia comandables, pero que no tienen implementado el control para el flujo bidireccional aún. También hay OBC que utilizan diodos, por lo que no les sería posible implementar la bidireccionalidad.

Cuando se apruebe el protocolo de comunicación bidireccional (segunda versión del ISO/IEC 15118-2), entonces los vehículos que tengan OBC bidireccional podrán comunicarse con los SAVE que soporten dicho protocolo. En cuanto al SAVE en AC: ni unidireccional, ni el bidireccional, utilizan conversores en su interior. Pero se diferencian en el protocolo de comunicación que soportan: IEC 61851-1 para unidireccional y ISO 15118-20 para bidireccional.

En este capítulo, se pondrá foco en el SAVE de corriente alterna, pues la idea es sugerir cambios al SAVE unidireccional ya diseñado, para que se pueda aplicar la bidireccionalidad.

[1]Si ya era más complejo implementar un SAVE unidireccional DC en comparación con el AC, se puede apreciar que en el SAVE bidireccional la diferencia de complejidad es mayor Pero un beneficio de tener el conversor bidireccional dentro del SAVE es que para una funcionalidad V2X, se pueden regular los parámetros dentro del SAVE, para cumplir con las normativas nacionales de la forma de onda inyectada a la red. Evitando así necesidad de acordar normas internacionales para que todos los OBC de los vehículos cumplan la regulación (ver [21])

10.5. Tendencia Global de Conversores OBC.

Según Paper [20] en abril 2019.

En la industria existe una tendencia global de aumentar la potencia en los conversores OBC. Es notoria, porque los primeros vehículos elétricos (VE) se cargaron a niveles de potencia de 3.3kW, pero actualmente casi todos los VE tienen OBC que soportan 6.6kW, 7.4 kW o 11 kW; y para potencias mayores hay adaptabilidad para cargas "Off-board".

El principal motivo de esta tendencia, es el aumento en la capacidad de la batería en cada VE, que a su vez, cumple el objetivo de lograr una mayor autonomía del VE y mejorar otras capacidades.

Actualmente la carga en alta potencia se realiza, mayoritariamente, mediante conversores externos ("Off-Board") en DC, pero a medida que haya más OBC de alta potencia en los VE, el consumidor se verá beneficiado. Porque en ese caso, habrá mayor oferta de puestos de carga, ya que las empresas de servicios públicos sólo tendrán que ofrecer SAVE de alta potencia que suministren en AC y así reducen costos, comparado con tener que instalar SAVE de carga rápida en DC.

Igualmente, mientras aumentan los niveles de potencia de los OBC, se presentan nuevos desafíos para lograr que los OBC sean compactos, eficientes y livianos; todas características requeridas para que el OBC pueda adaptarse junto a otros componentes dentro del VE.

Además de reducir los tiempos de carga y reducir costos de infraestructura para SAVEs de carga rápida, el aumento del nivel de potencia de los OBC también ayudará a las compañías de servicios públicos a que puedan incorporar funcionalidades de gestión inteligente de la curva de demanda. Por ejemplo, los OBC bidireccionales podrán utilizarse para funcionalidades como la reducción de picos de demanda y regulación de frecuencia (V2G) o suministrar energía eléctrica desde el vehículo al hogar (V2H) durante los cortes de suministro de la red.

10.5.1. Componentes Típicos a Bordo del VE.

Como se muestra en la Fig.10.1, los principales componentes de electrónica de potencia a bordo de un VE incluyen: el OBC, el módulo de potencia auxiliar (APM), el inversor de accionamiento del motor (*Motor Drive* inverter) y opcionalmente un conversor inalámbrico.

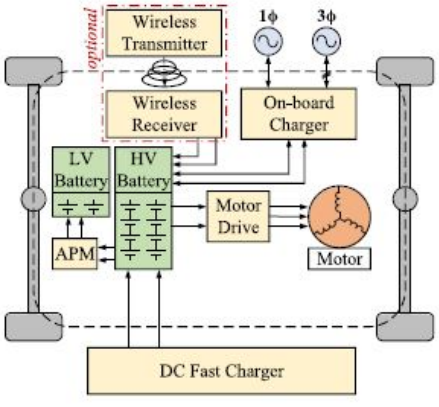


Figura 10.1: Componentes tipicos a bordo del VE.

Lo que posibilita los distintos métodos de carga de las baterías: la carga “On-Board” – donde se utiliza el OBC y el AMP -, la carga “Off-Board” y la carga inalámbrica, que a su vez, pueden clasificarse como cargas conductivas (“On-Board” y “Off-Board”) o cargas inductivas (inalámbrica).

El sistema de carga inalámbrica se ha representado como una interfaz opcional, porque aún no se han implementado en los vehículos eléctricos. Si bien los futuros vehículos eléctricos, tal vez solo tengan interfaces de carga conductivas o solo inductivas, es posible que puedan tener ambas interfaces para la interoperabilidad y la compatibilidad de carga.

En este resumen, del estudio citado, se presentan diversas estructuras de OBC investigadas para cargas “On-Board” de alta potencia (mayores a 7,4kW).

10.5.2. Características de los OBC.

Los OBC están diseñados como unidades independientes que se comunican con un SAVE y con el sistema de gestión de baterías (BMS) del vehículo, para entregar el voltaje y la corriente de carga solicitados a la batería del VE. Por lo tanto, una parte importante del proceso de carga es que el BMS interprete correctamente el estado de carga (SoC) de la batería y que el OBC entregue el perfil de carga correcto.

Además el OBC, en modo de carga, debe garantizar que se cumplan los estándares de calidad de energía en el lado de la red. Los estándares estadounidenses, europeos y chinos con respecto a la inyección armónica para estos dispositivos en modo de carga, se abordan en SAE J2894, IEC 61000, (GB/T) 14549 respectivamente, y en IEEE 519. Mientras que el aislamiento galvánico es un requisito en la mayoría de los sistemas OBC de VE según las normas de seguridad incluidas las UL 2202 e IEC 60950.

Por otro lado, si los VE funcionan como fuente y el OBC se utiliza para regular el flujo de energía externo, el OBC debe cumplir con los estándares del inversor de micro-generación, detallados en IEEE 1547, IEEE 2030, IEC 62109, UL 1741 y NB/T 33015. A medida que los modos de operación con respecto a los vehículos eléctricos como fuente de energía se vuelven más maduros, puede ser necesario plantear estándares más estrictos y bien definidos.

Los OBC de VE se clasifican según sus niveles de potencia:

Clasificación	nivel de potencia	Conector EEUU	Conector UE	Conector China
AC nivel 1	menor a 3.7kW	SAE J1772	N/A	N/A
AC nivel 2	entre 3.7kW - 22kW	SAE J1772 o Tesla	IEC62196-2	GB/T 20234 AC
AC nivel 3	entre 22kW - 43.5kW	SAE J3068	IEC62196-2	GB/T 20234 AC
DC nivel 3	menor a 200kW	CCS Combo1 CHAdeMO/Tesla	CCS Combo2 CHAdeMO/Tesla	GB/T 20234 DC CHAdeMO/Tesla

Tabla 10.1: Clasificación según nivel de potencia y los tipos de conectores correspondientes.

10.5.3. OBC Unidireccionales y Bidireccionales.

Los sistemas OBC pueden soportar un flujo de potencia activa unidireccional o bidireccional, como muestra la Fig.10.2, cada uno con distintas ventajas y desventajas.

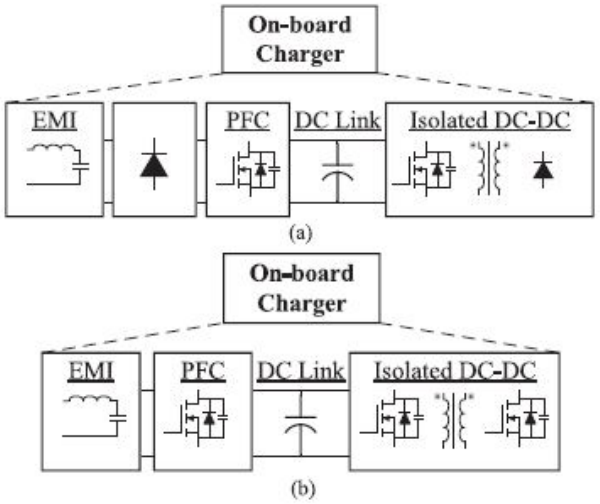


Figura 10.2: (a) OBC unidireccional.; (b) OBC bidireccional.

Si bien algunos OBC actuales pueden ser compatibles con el flujo de potencia activa bidireccional, esta característica no se ha utilizado, ya que las empresas de servicios públicos aún no tienen SAVE que puedan implementar la comunicación para dicha funcionalidad. Y a pesar de que implicaría un mayor costo, un aumento de volumen de OBC y una degradación de la batería al utilizarlo, se cree que en el futuro, los OBC bidireccionales serán más frecuentes

La estructura típica de un OBC unidireccional se muestra en la Fig.10.2(a). Las ventajas de los OBC unidireccionales son la reducción de la complejidad del sistema, la menor cantidad de componentes activos, por lo tanto, menor costo y con un tamaño compacto. Sin embargo, la mayor desventaja radica en la incapacidad de adaptarse a las futuras funcionalidades de gestión inteligente de la demanda.

Estos OBC reducen complejidad utilizando puentes de diodos, tanto en el lado frontal (front-end) como en el lado secundario (después del aislamiento). Igualmente, hay otros modelos de OBC unidireccionales que usan al menos un front-end activo, para la Compensador del Factor de Potencia (PFC). Esta etapa presenta beneficios como: un control del factor de potencia ajustable y una mayor eficiencia debido a una reducción de componentes en el resto del circuito PFC.

Los OBC bidireccionales tienen una estructura típica como la que se muestra en la Fig.10.2(b).

La principal ventaja de los OBC bidireccionales, es que permiten el flujo de potencia activa tanto desde la red hacia el vehículo (G2V), como también desde el vehículo hacia el hogar (V2H) o hacia una carga externa (V2L) o hacia la red eléctrica (V2G).

Para dichos OBC se necesita tanto un front-end activo con PFC, como también un lado secundario DC-DC activo. Como consecuencia tiene que utilizar mayor cantidad de componentes sin perder confiabilidad, por lo que aumenta el costo y el peso del sistema.

10.5.4. Sistemas OBC No-Integrados o Integrados.

Un sistema OBC no-integrado, es cuando el OBC es una unidad independiente dentro del VE, con la única funcionalidad de acondicionar adecuadamente el voltaje y corriente de su salida para cargar la batería del VE a partir de la corriente alterna suministrada por el SAVE. Si fuese un OBC bidireccional, el sentido del flujo de potencia activa, también puede ser el inverso. En cambio, los sistemas OBC integrados, tienen funciones de electrónica de potencia adicionales que se integran con el OBC.

10.5.5. OBC No-Integrados.

La estructura típica de un OBC no-integrado incluye un filtro de EMI (interferencia electromagnética), un compensador de factor de potencia AC/DC, un enlace DC y un convertidor DC/DC aislado.

La principal dificultad de los OBC de alta potencia es equilibrar la eficiencia, el costo, el tamaño, la confiabilidad y el peso. Cuanto mayor es el nivel de potencia, más considerables son las perdidas de conducción asociadas y por lo tanto, pueden requerir dispositivos de conmutación en paralelo. Cada uno de los diseños presentados, propone una estrategia particular para resolver dicha problemática. La mayoría de las topologías son de dos etapas, aunque también hay algunas de una etapa.

- **1) Modular de dos Etapas I:** En la figura 10.3(a) se muestra un modulo (de 3,5 kW) de una fase del OBC trifásico (de 10,5 kW). Cada módulo monofásico se compone de un puente de diodos y un convertidor boost operando a una frecuencia de switcheo de 90 kHz, conectado a un enlace DC. En la entrada de cada modulo, hay un único filtro EMI. Luego, la etapa DC/DC es un half-bridge aislado, con un puente de diodos en el secundario.

  Este OBC alcanza una eficiencia del $95,6\,\%$ y al tener puentes de diodos, reduce su complejidad. Además tiene la ventaja de que un convertidor LLC half-bridge tiene switches que conmutan con cero voltaje (ZVS).

  Mientras que las desventajas son el peso y tamaño del sistema, además de problemas de confiabilidad en el uso del enlace DC y alto estrés en los condensadores resonantes. Es un conversor que solo permite un flujo de potencia unidireccional.

- **2) Modular de dos Etapas II:** En la figura 10.3(b) se muestra un módulo (de 7,4 kW) de una fase del OBC trifásico (de 22 kW para red europea). Cada módulo monofásico se compone de un puente de diodos con un limitador de corriente inrush y un convertidor boost con dos ramas intercalables conectadas a un enlace DC. En la entrada de cada módulo, hay un único filtro EMI.

  Luego, la etapa DC/DC es con dos full-bridge aislados (con entradas y salidas en paralelo) con un puente de diodos en el secundario.

  Este OBC alcanza una eficiencia del $95,6\,\%$ y al tener puentes de diodos, reduce su complejidad. Las ventajas incluyen la disminución del ripple de la corriente de salida y el tamaño compacto del cargador.

  Las desventajas incluyen que el flujo de energía es unidireccional y la gran cantidad de componentes del circuito, aunque solo diez son de conmutación activa.

- **3) Modular de una Etapa I:** En la figura 10.3(c) se muestra un módulo (de 7,2 kW) de una fase del OBC trifásico (de 22 kW). Cada módulo monofásico se compone de un rectificador full-bridge (R1-R4). Luego hay un enlace AC y le sigue un convertidor DC/DC full-bridge aislado con todos los componentes activos.

  Este OBC alcanza una eficiencia máxima de más de 97 % y una alta densidad de potencia (potencia por unidad de volumen del OBC), sin incluir un condensador de enlace en DC. La mayor ventaja es que se puede implementar un flujo de potencia bidireccional, pero al tener mayor cantidad de componentes activos, el algoritmo de control es más complejo y el costo del OBC aumenta.

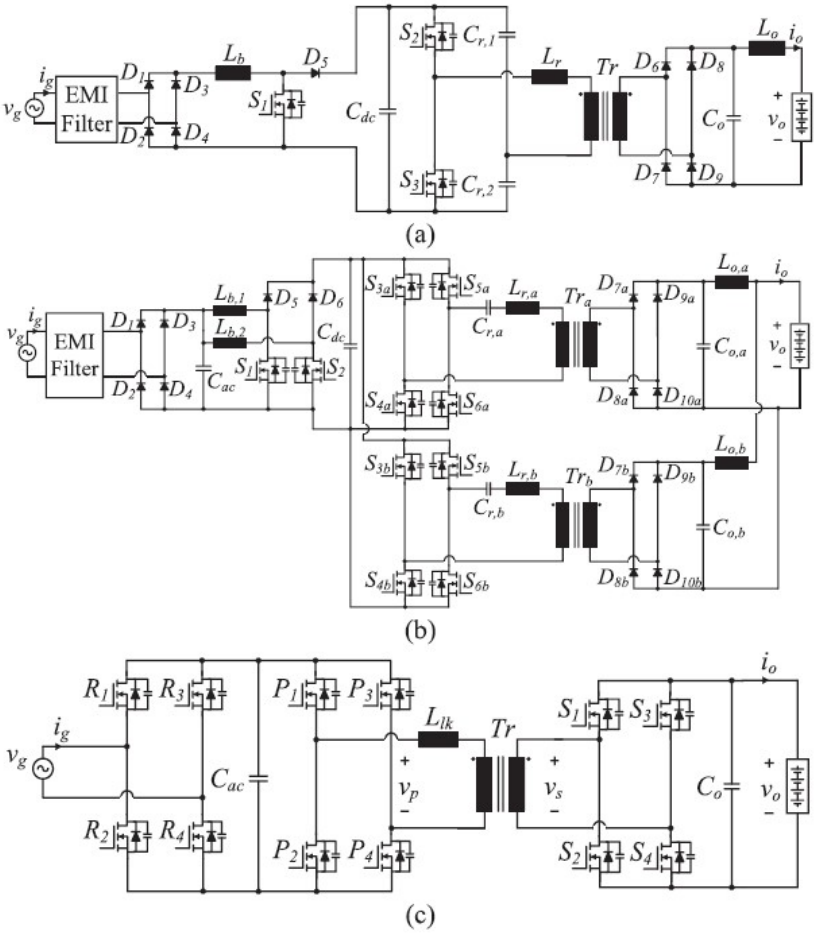


Figura 10.3: (a) Modular de dos etapas I.; (b) Modular de dos etapas II.; (c) Modular de una etapa I.

- **4) Full-Power de dos Etapas I:** En la figura 10.4(a) se muestra un OBC con entrada trifásica de 10 kW de potencia nominal. Se compone de un PFC trifásico conectado a un condensador de enlace DC, seguido por un convertidor aislado LCC resonante: con un half-bridge en el primario y un full-bridge en el secundario.

  Este OBC alcanza una eficiencia del 96 % y tiene la ventaja de un flujo de potencia bidireccional, debido a que tiene partes activas del lado primario y del secundario.

  Sin embargo, las desventajas radican en las altas corrientes RMS del half-bridge LLC, la confiabilidad debido al mayor estres en los condensadores resonantes y el requisito de un gran filtro EMI.

- **5) Full-Power de dos Etapas II:** En la figura 10.4(b) se muestra un OBC con entrada trifásica de 20 kW de potencia nominal. Se compone de un boost PFC trifásico conectado a un condensador de enlace DC, seguido por dos convertidores LCC resonantes, aislados y conectados en paralelo, cada uno con un half-bridge en el primario, con transformadores de alta frecuencia (conectados en estrella) y cada uno con un puente de diodos en el secundario.

  Este OBC alcanza una eficiencia del 96 % y tiene la ventaja de tener un rango flexible de voltaje de salida, una corriente balanceada entre los transformadores y un alto nivel de potencia con una eficiencia razonablemente alta.

  Las desventajas incluyen: corrientes RMS potencialmente altas (a 20 kW) a través del Half-bridge LLC, problemas de confiabilidad en el enlace de DC y condensadores resonantes. Y permite únicamente flujo de potencia unidireccional debido al uso de puentes de diodos secundarios. Mientras que la bidireccionalidad podría lograrse, utilizando MOSFET en un full-bridge del lado secundario, a costa de una mayor complejidad del sistema.

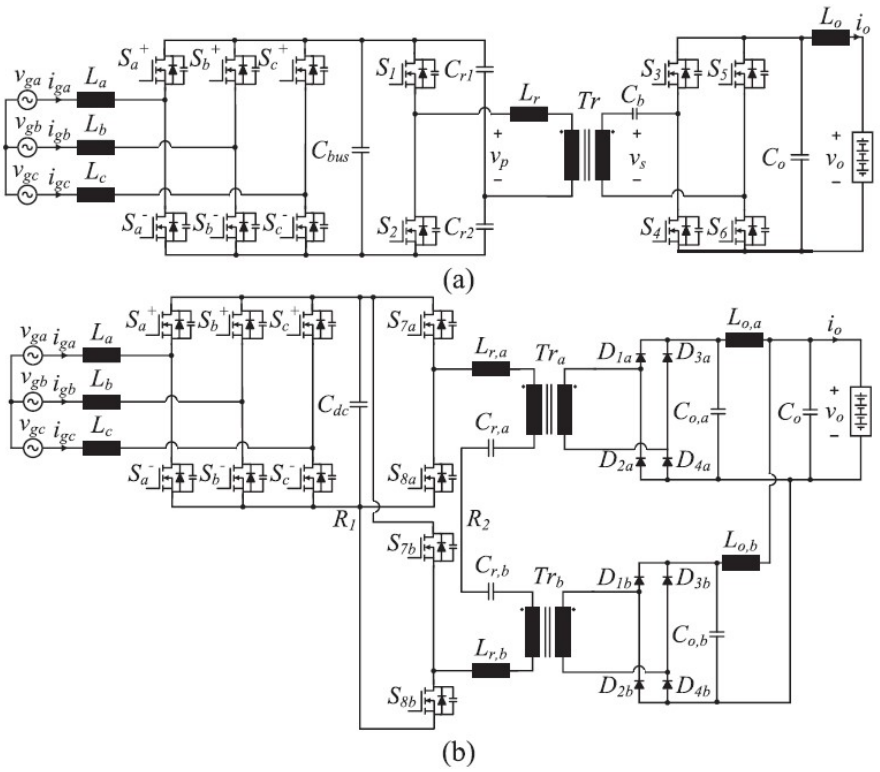


Figura 10.4: (a) Full-Power de dos etapas I.; (b) Full-Power de dos etapas II.

- **6) Full-Power de una Etapa I:** En la figura 10.5(a) se muestra un OBC con entrada trifásica de 10 kW de potencia nominal. Se compone de un convertidor matrix de tres fases (con switches back-to-back), un enlace inductor y un transformador de alta frecuencia. Seguido por un full-bridge activo en el lado secundario. Debido a que todavía no fue construido, no se pudieron realizar ensayos para medir su eficiencia.

  Entre las ventajas de este OBC, está el hecho de que permite el flujo de potencia bidireccional y la compensación de reactiva en la entrada trifásica. También, al ser un conversor de una sola etapa, se obtiene una alta densidad de potencia (potencia por unidad de volumen del OBC).

  Las desventajas incluyen una implementacion de control compleja, una compatibilidad desafianta con una entrada monofásica y una alta cantidad de semiconductores debido a la necesidad de switches bidireccionales en el lado primario.

- **7) Full-Power de una Etapa II:** En la figura 10.5(b) se muestra un OBC con entrada trifásica de 11 kW de potencia nominal. Se compone de tres puertos AC, cada uno con un circuito tipo-T, transformadores de alta frecuencia y un full-bridge activo en el lado secundario. Los circuitos tipo-T conectan las tres fases AC de la entrada, con los tres bobinados del transformador que están en el lado primario y además estos circuitos se conectan entre si en estrella.

  Este OBC alcanza una eficiencia del 96 % y tiene las ventajas de un flujo de potencia bidireccional, un diseño compacto y un condensador de menor valor por la cancelación del ripple.

Las desventajas incluyen una alta cantidad de semiconductores del lado primario, un esquema de control complejo y una implementación monofásica difícil.

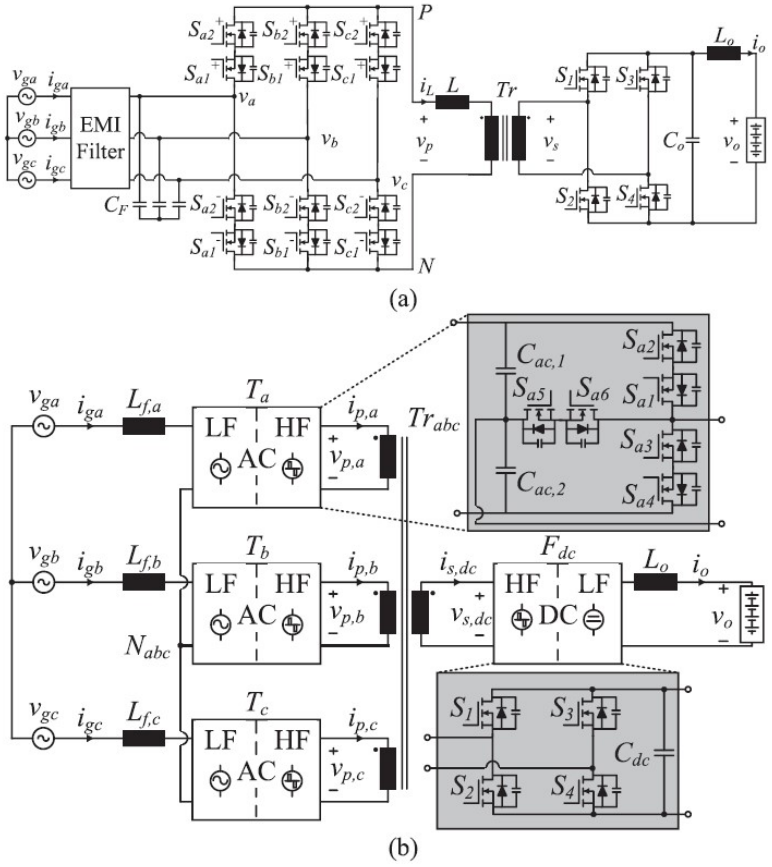


Figura 10.5: (a) Full-Power de una etapa I. ; (b) Full-Power de una etapa II.

A continuación se muestra un resumen de los conversores, On-Board no-integrados, analizados anteriormente:

¿Bidireccional?	Figura	Modelo	kW	Eficiencia	Topología
No	10.3.(a)	"Modular OBC" de dos etapas I	10.5	95,6 %	[33]
No	10.3.(b)	"Modular OBC" de dos etapas II	22	94,5 %	[34]
Si	10.3.(c)	"Modular OBC"de una etapa I	22	> 97 %	[35]
Si	10.4.(a)	"Full-Power OBC" de dos etapas I	10	96 %	[36]
No	10.4.(b)	"Full-Power OBC" de dos etapas II	20	96 %	[37]
Si	10.5.(a)	"Full-Power OBC" de una etapa I	10	−−	[38]
Si	10.5.(b)	"Full-Power OBC" de una etapa II	11	96 %	[39]

Tabla 10.2: Lista de los conversores On-Board no-integrados analizados.

10.5.6. Sistema OBC Integrado.

Los sistemas OBC integrados, además de ejecutar la carga "On-Board'" se combina con otros componentes existentes y necesarios dentro del VE. Lo más común para los OBC de alta potencia es que estén combinados con el sistema de propulsión, que consiste en el motor eléctrico y el inversor (*motor drive*).

La combinación mencionada, incluye todos los componentes necesarios en un sistema OBC típico (interruptores, diodos, inductores y condensadores). Por lo tanto, con la adición de una interfaz frontal entre red-motor o la posible reconfiguración de los devanados del motor, se puede realizar una carga de alta potencia sin generar un par en el motor.

Los motores VE típicos tienen una potencia entre 30 kW y 200 kW, lo que establece la posibilidad de que los sistemas OBC integrados puedan competir con los sistemas de carga ultra-rápida en DC "Off-Board". Sin embargo, una desventaja de la integración del OBC al sistema de propulsión, podría ser un mayor estrés en el sistema de propulsión.

La Fig. 10.6(a) es el diagrama de bloques de un sistema de propulsión típico OBC integrado.

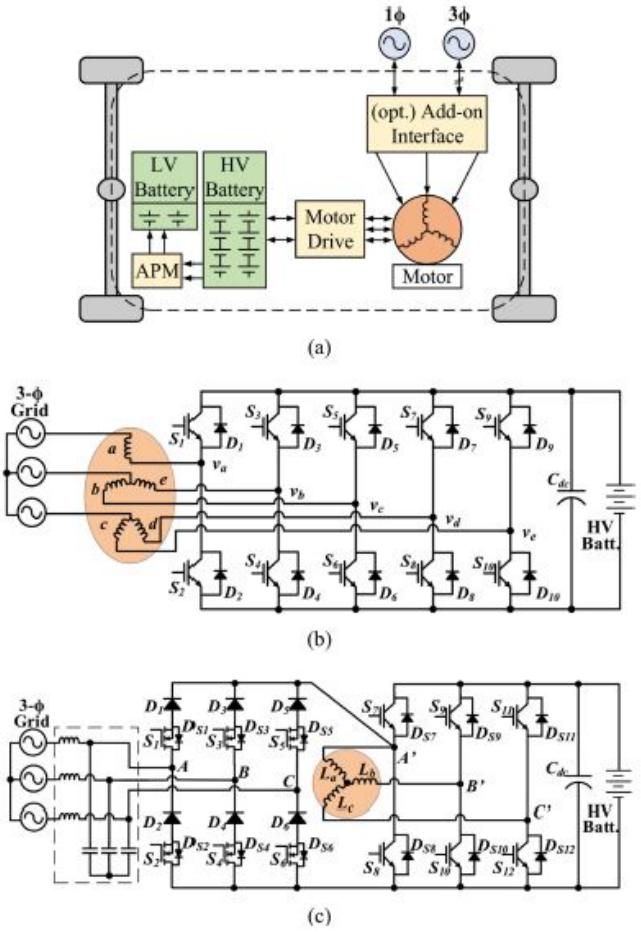


Figura 10.6: OBC integrado. (a) diagrama de bloques. (b) con sistema de propulsión polifásico. (c) sin modificaciones al sistema de propulsión trifásico.

Los sistemas OBC integrados al sistema de propulsión, se pueden clasificar en cinco categorías según los requisitos de la interfaz: acceso al punto neutro del motor, bobinados de motor dividido, reconfiguración de bobinados de motor, sistema polifásico de propulsión y sistemas de interfaz complementarios.

La mayoría de las diferentes estructuras de este tipo de OBC integrados, requiere un sistema de propulsión modificado o una máquina eléctrica diseñada a medida para que sea adecuada para el modo de operación de carga.

Aquí se presentan dos OBC integrados por propulsión:

- **1) Sistema Polifásico de Propulsión:** El motivo de este sistema OBC integrado es reducir la cantidad de componentes extra en el VE, evitando también que se genere torque en el motor durante la carga. En este sistema, se desacoplan las inductancias de la máquina de propulsión y se utilizan como filtro de entrada al inversor polifásico, el cual, durante la carga de la batería, termina siendo utilizado como un rectificador conectado directamente a la batería del vehículo. Se probó que la capacidad de flujo bidireccional de potencia, es posible, tanto con entradas de red trifásicas, como con monofásicas.

  Un ejemplo es el presentado en la figura 10.6(b) y un estudio completo de este tipo de sistemas y su algoritmo de control se puede encontrar en [40]. La principal desventaja de los sistemas integrados OBC de propulsión polifásica es que su implementación está limitada, debido a que actualmente dominan las máquinas de propulsión trifásicas.

- **2) Interfaz Añadida:** Teniendo en cuenta el caso anterior, es una ventaja si un sistema OBC integrado puede utilizar una típica máquina de propulsión trifásica sin modificaciones (ver [41]). En el ejemplo presentado en la figura 10.6(c), se agrega un conversor y un filtro EMI, entre la entrada trifásica en AC y las tres inductancias de la máquina de propulsión. Mientras que el inversor entre la batería y la máquina de propulsión, no sufre modificaciones. Para este tipo de sistemas OBC, aún no existe un control para el flujo bidireccional de potencia.

10.5.7. Capacidades Futuras de los OBC.

En la medida en que se utilice mayor cantidad de vehículos eléctricos, las empresas de servicios públicos buscarán utilizar funcionalidades de gestión inteligente que los OBC puedan ejecutar, por ejemplo:

- **1) Peak Shaving:**

  La modalidad V2G permite que la batería del VE se comporte como una fuente de almacenamiento, que suministra energía eléctrica a la red cuando lo requiera la empresa de servicios públicos, que generalmente será durante el período de alta demanda de energía eléctrica.

  Si esta funcionalidad es ejecutada a gran escala, los picos de demanda se reducirán, pues los VE dejarán de consumir durante ese período y además podrán suministrar la energía eléctrica que almacenaron en períodos de baja demanda (ver [27]).

  Una de las limitaciones en el "Peak Shaving" es el aumento de la degradación de la batería, ya que la potencia activa se transfiere tanto en el modo de operación G2V, como en V2G (ver [28]). Y cuanto mayor sea la profundidad de la descarga requerida por la empresa de servicios públicos, mayor será la degradación de la batería.

- **2) Servicios Auxiliares:**

  Aunque los modos G2V y V2G sean los predominantes para los OBC, también existen otras operaciones de gestión inteligentes. Por ejemplo, el modo V2H, donde el VE se utiliza como fuente de alimentación para un hogar aislado o como una UPS *offline* que se utiliza para cuando un hogar se desconecta de la red. Y existe el modo *Vehicle-for-Grid* (V4G), donde el VE proporciona funciones de soporte de red, como compensación de potencia reactiva y regulación de frecuencia (ver [29]).

  Para el modo V4G, se estudió un OBC no-aislado de dos etapas (ver [30]), que teniendo un control de potencia activa y reactiva simultaneo, pudo compensar reactiva sin afectar la vida útil de la bateria y sin afectar el SoC. Con los OBC aislados de dos etapas se llegaría a las mismas conclusiones, ya que la etapa PFC frontal puede asumir la responsabilidad exclusiva de generar un voltaje de DC-link variable con el factor de potencia solicitado (ver [31]). Una desventaja es que este OBC de dos etapas, tendrá una mayor carga en el condensador DC-link, ya que habrá más ciclos de carga y descarga, en comparación con el control de potencia activa únicamente.

  Se utilizan también OBC para la regulación de frecuencia de la red, como un servicio de soporte más rápido que los sistemas mecánicos utilizados actualmente, si se controlan adecuadamente. Hay análisis detallados sobre los OBC que lo proporcionan y su posible impacto sobre la degradación de la batería del VE (ver [32]).

  Según una visión de costo-beneficio, pareciera que no alcanza solo con pagarle a los propietarios de VE por el servicio prestado a la red. Seguramente se tendrá que pensar en otros incentivos por parte del gobierno y de las empresas de servicio público.

10.5.8. Resumen.

Se han identificado varias tendencias claves que podrían dar forma al futuro de las soluciones de carga "On-Board"de VE:

- En futuros VE habrá una mayor capacidad de la batería y mayores voltajes de la batería, en la medida en que los fabricantes de automóviles se vean impulsados por las iniciativas gubernamentales para reducir las emisiones y haya una demanda por la necesidad de aumentar el rango de alcance de los VE y mejorar otras capacidades.
- El aumento del nivel de potencia de los OBC durante la carga, será un requisito en los futuros VE, para mejorar los tiempos de carga durante la noche, para VE de mayor capacidad de la batería. Y facilitar una infraestructura para una carga más rápida, sin instalaciones masivas de SAVEs de carga rápida de DC de nivel 3.
- Los OBC integrados, incluida la máquina de propulsión, los APM y los conversores inalámbricos integrados, son enfoques prometedores para aumentar la potencia nominal del OBC, al tiempo que evitan un impacto significativo en el peso y el volumen del sistema de carga.
- Con el aumento del nivel de potencia de los OBC, la carga deberá coordinarse y controlarse. Además, los OBC deberán ser bidireccionales en el futuro y compatibles con funcionalidades de gestión inteligente de la demanda en la red eléctrica. Esto no solo proporcionará funcionalidades adicionales a los OBC para utilidades como el "aplanamiento de la curva de demanda" en V2G, sino también para que los propietarios de VE utilicen la energía almacenada de su VE, por ejemplo, en su hogar.

10.5.9. Conclusiones de la Tendencia Global, Según Estudio Citado.

La tendencia de la industria a aumentar el nivel de potencia de los OBC, es evidente, por el hecho de que la cantidad de OBC de nivel 2 ya se han duplicado en los últimos cinco años.

Con este fin, se han presentado y comparado OBC no integrados de alta potencia, incluidas las soluciones de carga de dos etapas de procesamiento de potencia modular y de potencia completa (que utilizan un condensador de enlace DC) y soluciones de carga de una sola etapa que no utilizan un condensador de enlace DC.

Aparte de los aumentos en los niveles de potencia de OBC, la coordinación de carga desde el lado de la empresa de servicios públicos, y de los SAVEs, se convertirá en una necesidad para preservar el rendimiento dinámico de la red eléctrica durante la adopción generalizada de VE. Además, las funcionalidades de los OBC se expandirán e incluirán servicios de red auxiliares, junto con un flujo de energía V2X controlado.

10.6. Protocolos de Comunicación Entre SAVE y VE.

Como se mencionó en capítulos anteriores, la comunicación entre el SAVE y el VE está definida en el Anexo A del protocolo IEC 61851-1 y es idéntica a la establecida en SAE J1772. Dichos protocolos establecen que con una señal PWM (desde $-12\ V$ hasta $12\ V$ o $9\ V$ o $6\ V$), el SAVE varía el Duty-Cycle para comunicar la corriente máxima que va a entregar y el VE varía el voltaje máximo para comunicar en que estado de carga (SoC) se encuentra. Para generar esa señal, fue necesario diseñar el circuito descripto en el capítulo 6.

El problema es que el protocolo IEC 61581-1 menciona claramente que no considera requerimientos para la transferencia bidireccional de energía: "Requirements for bi-directional energy transfer are under consideration and are not in this edition" of IEC 61851-1: 2017 (edition 3.0).

Para habilitar la transferencia bidireccional de energía, se podrá aplicar la segunda versión del protocolo ISO/IEC 15118-2 [2], que se espera que se publique a fines del año 2021. La primera versión de dicho protocolo ya fue aprobada y está siendo implementada en los vehículos como el modelo Audi e-tron (o Porsche Taycan o Smart Electric Drive desde 2017) utilizando los conectores de corriente continua CCS-combo.

Esta primera versión del protocolo habilita una comunicación tipo cliente-servidor, en la cual el VE es el cliente y el SAVE es el servidor. El VE, envía un mensaje de solicitud de datos (**request**) y el SAVE responderá (**response**) con los datos correspondientes. Pero no se puede realizar una descarga de la batería del VE, porque aún no se publicó la segunda versión del protocolo que establece los mensajes para una descarga de la batería (V2X).

10.7. Comparación Entre ISO 15118 y IEC 61851-1.

El protocolo ISO 15118 se basa en el IEC 61851-1, es decir, el SAVE también genera una señal PWM para que el VE comunique su estado de carga; por lo que los estados A, B, C, D, E y F siguen siendo definidos de la misma manera (ver capítulo 5). Entonces, si el SAVE (o VE) sólo implementa el IEC 61851, igual será compatible con cualquier VE (o SAVE) que implemente el ISO 15118.

La diferencia es que, si el SAVE soporta el protocolo ISO 15118, el Duty-Cycle del PWM que genera el SAVE será del 5 % para solicitar una comunicación PLC con el vehículo. Si el VE también soporta este protocolo, entonces a partir de ese momento se comunicarán mediante PLC (Power Line Communication) en el pin Control Pilot (CP). De lo contrario, si el VE solo soporta el protocolo IEC 61581-1, entonces el SAVE modificará el Duty-Cycle entre 10 % hasta 96 % para indicar la mayor corriente que le suministrará (como lo indica dicho protocolo).

[2] Para mayor información, se recomienda visitar el sitio web `https://v2g-clarity.com/`

La funcionalidad distintiva de la primera versión del protocolo ISO/IEC 15118 es que habilita una comunicación de alto nivel, en la que los mensajes del VE y del SAVE pueden tener mayor cantidad de datos e intercambiarlos de forma segura. Por ejemplo, en la funcionalidad **Plug&Charge** permite la autenticación del vehículo, apenas se lo conecte al SAVE, sin necesidad de que el usuario pase una tarjeta. Dicha información es la que transmite mediante PLC en el pin CP.

Sobre PLC (Power Line Comunication):

"El PLC es como cualquier otra tecnología de comunicación mediante la cual un emisor modula los datos que se enviarán, los inyecta en un medio y el receptor desmodula los datos para leerlos. La principal diferencia es que el PLC no necesita cableado adicional, reutiliza el cableado existente." [ver [43]]

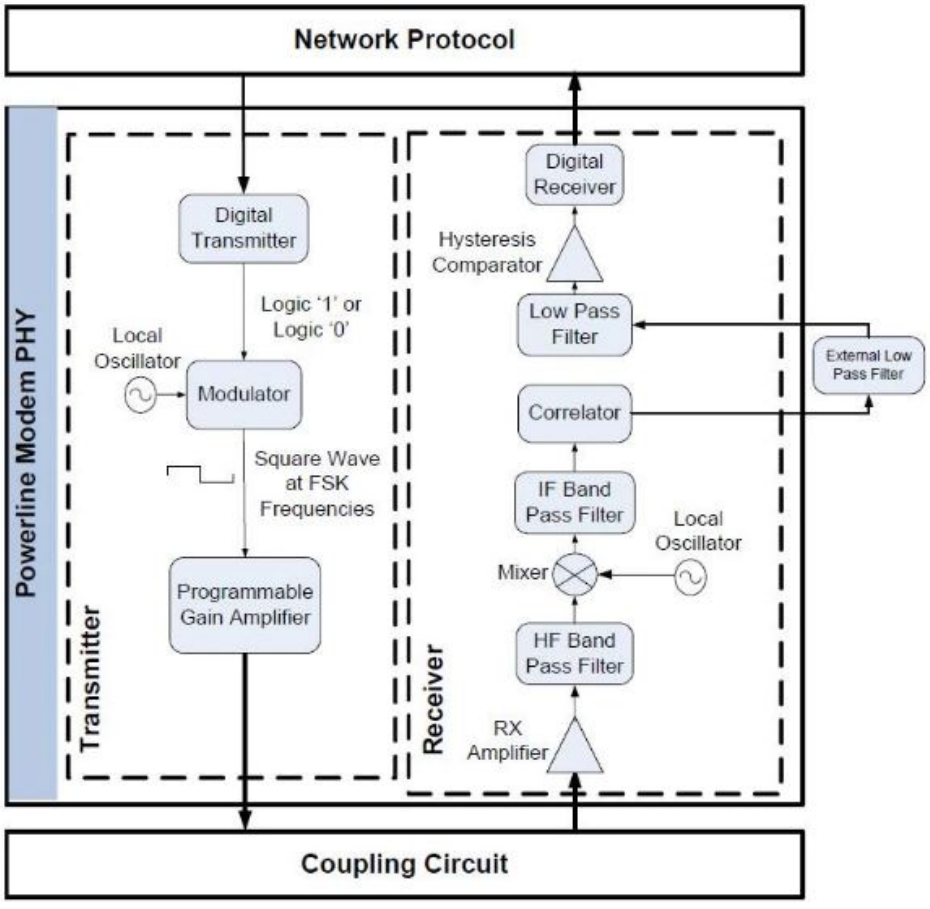


Figura 10.7: Diagrama de bloques indicando lo necesario para implementar una Power Line Comunication (PLC)

El SAVE tiene una unidad de control denominada SECC (Supply Equipment Communication Controller), que entre otras funciones tiene que codificar los mensajes que son enviados por el SAVE y para descodificar los mensajes del VE, y análogamente, el VE tiene una unidad de control denominada EVCC (Electric Vehicle Communication Controller). Ambas unidades

tienen que asegurarse de ejecutar correctamente el protocolo ISO 15118.

10.8. Tendencia en Conectores y Entradas al VE.

IEC y SAE acordaron en universalizar conectores: monofásicos AC, trifásicos AC y ultrarápidos en DC (ver CharIN [42]), por lo que la tendencia -en los países que apliquen esas normas- será utilizar los conectores Combined Charging System (CCS).

Actualmente, los conectores y las entradas a VE, que cumplen la especificación CCS 2.0[3]:

- Para cargas en DC, requieren aplicar la primera versión del protocolo ISO/IEC 15118, con comunicación de alto nivel PLC.
- Mientras que para cargas AC, se aplica el mismo protocolo, pero existe también la posibilidad de comunicación sólo con PWM.

Cuando la segunda versión del protocolo ISO/IEC 15118-2 defina la carga bidireccional, es esperable que la especificación CCS 3.0 la implemente. Mientras tanto, los conectores CCS y entradas a vehículos eléctricos CCS, solo aplican a cargas unidireccionales.

Figura 10.8: Tipo de conectores CCS.

Como fue explicado en el capitulo 2 y según muestra la imagen anterior:

Si un VE se conecta mediante un CCS-combo2 macho, podrá cargar, en AC o en DC.

Y se cargará en AC, si el SAVE tiene un conector tipo 2 hembra.

Y se cargará en DC, si el SAVE tiene un conector CCS-combo2 hembra.

Se observa entonces que para implementar la comunicación de alto nivel establecida en el protocolo ISO/IEC 15118-2 y en un futuro implementar la descarga de una batería (V2X), se debe utilizar un conector que cumpla con la especificación CCS 2.0 o superior.

[3]Aclaración: la CCS 2.0 aplica la ISO/IEC 15118 y la utilizan los VE diseñados a partir del 2015. Pero la CCS 1.0 no aplicaba dicho protocolo, sino el alemán DIN SPEC 70121:2014-12).

10.9. Sobre el Protocolo ISO/IEC 15118.

Este protocolo define los casos de uso (en la primera parte) y la interfaz de comunicación entre el VE y el SAVE bajo el modelo de siete capas de la Open System Interconnection (OSI), desde la capa de aplicación hasta la capa física (segunda y tercera parte). El protocolo además adoptó el estandard HomePlug Green PHY (HPGP) para PLC en el medio físico de comunicación (ver paper [23]).

Como se observa en la figura 10.9 las capas físicas y de enlace de datos se definen en la Parte 3 y todas las capas superiores se definen en la Parte 2.

Figura 10.9: Las siete capas OSI y la relación con las partes del protocolo.

En la capa física y la capa de enlace de datos, se utiliza Power Line Communication (PLC). En particular, se adopta el estándar HPGP para PLC, pero se podrían usar otros tipos de tecnologías de PLC, si HPGP no se puede usar en ciertos países. Para las pruebas, ISO/IEC 15118 Parte 5 solo se centra en el protocolo de emparejamiento de HPGP, denominado Signal Level Attenuation Characterization (SLAC).

En la capa superior del PLC, se usa IPv6 para la capa de red y UDP (User Datagram Protocol) / TCP (Transmission Control Protocol) para la capa de transporte.

Para la capa de gestíon de sesiones, ISO/IEC 15118 define Vehicle-to-Grid Transfer Protocol (V2GTP), y todos los mensajes se entregan en paquetes V2GTP. El protocolo V2GTP simplemente indica la versión de ISO/IEC 15118 en acción y el tipo de paquete incluido.

El primer tipo de paquete del paquete V2GTP es SDP (SECC Discovery Protocol). Mediante este protocolo, el EVCC transmite un paquete de solicitud SDP y espera una respuesta de un SECC. A partir de la respuesta, EVCC se entera de la dirección IP y el puerto de comunicación de la SECC.

El segundo tipo de paquete es el mensaje de la aplicación. Cada mensaje de la aplicación se representa internamente en XML (Extensible Markup Language) para una máxima flexibilidad y portabilidad, pero el mensaje real en tránsito está codificado por el algoritmo EXI (Efficient XML Interchange) para que el paquete sea compacto.

Esta primera versión del protocolo ISO/IEC 15118 habilita una comunicación tipo cliente-servidor, en la cual el VE es el cliente y el SAVE es el servidor. El VE, envía un mensaje de solicitud de datos (**request**) y el SAVE responderá (**response**) con los datos correspondientes. Mediante una serie de mensajes de **request** y **response**, el VE y el SAVE intercambian la información necesaria antes, durante y después de la carga.

El orden de los tipos de mensajes de la aplicación en la figura 10.9 coincide aproximadamente con el orden de comunicación real, pero no son necesariamente los mismos. Por seguridad, TLS (Transport Layer Security) ayuda al VE a autenticar al SAVE, protegiendo la confidencialidad e integridad de los mensajes de la aplicación. Las características de seguridad XML proporcionan protección adicional de información confidencial. Con el cifrado XML y las características de firma, el VE puede proteger la confidencialidad y autenticidad de la información confidencial, como los números de tarjetas de crédito. La prueba explícita de estas funcionalidades está fuera del alcance de este trabajo.

Home Plug Green PHY: Physical/Data-Link Layer.

Como medio de comunicación, ISO/IEC 15118 recomienda Home Plug Green PHY (HPGP). En particular, su mecanismo de emparejamiento SLAC es un paso crucial en la conexión VE-SAVE. En este mecanismo, el SAVE mide y calcula el nivel de señal de VE para determinar cuál VE está conectado con él a través del enchufe de carga (ver [23]).

10.10. Procedimiento para Implementar ISO/IEC 15118.

En la figura 10.10 se observa un resumen del procedimiento de carga de un VE, según lo establecido en la primera versión del protocolo ISO 15118 (ver [23]). Recordar que la primera versión del protocolo no habilita la transferencia de energía eléctrica bidireccional.

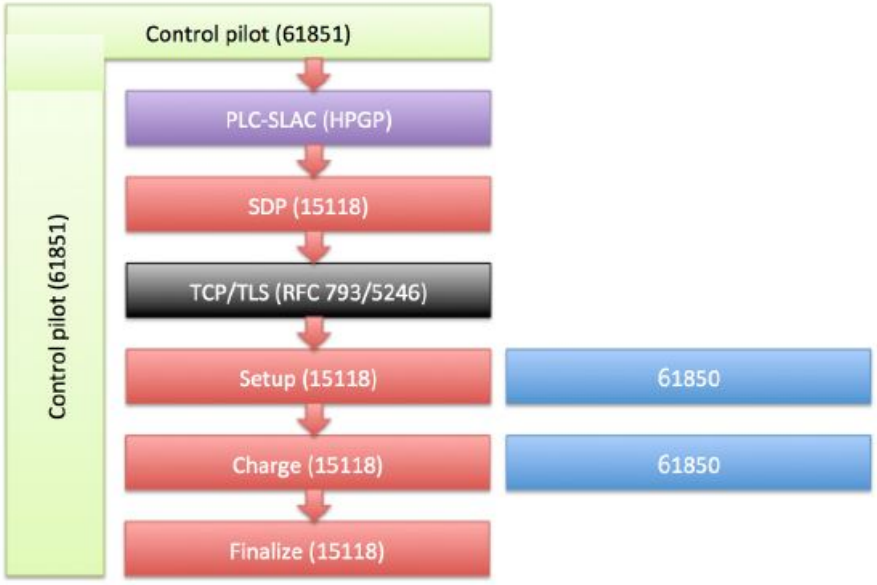


Figura 10.10: Resumen del procedimiento de carga, indicando los estándares relevantes.

- **Cuando el usuario coloca el conector:** el VE y el SAVE detectan el flujo de corriente e inician el protocolo IEC 61851 en el Control Pilot (CP). De esa manera, ambas partes pueden reconocer diferentes estados a lo largo del proceso de carga.
- **Una vez que VE y SAVE están listos:** inician una conexión PLC mediante el protocolo SLAC como se define en HPGP.
- **Una vez que se establece la conexión del PLC:** el VE emite un paquete SDP (SECC Descovery Protocol) en la Red de Area Local (LAN) para identificar la dirección IP y el número de puerto del SAVE.
- **Después de recibir dicha información:** el VE realiza una conexión TCP (Transmission Control Protocol) al SAVE, asegurado por TLS (Transport Layer Security).

A partir de esta situación, el VE y el SAVE pueden intercambiar paquetes TCP de forma segura para controlar el procedimiento de carga. Se puede dividir el procedimiento en tres fases: configuración, carga y finalización.

En la fase de configuración, el VE y el SAVE negocian varios parámetros, incluida la versión del protocolo, la identidad del usuario, la información del servicio, los métodos de pago y los parámetros de cobro. En la fase de carga, realizan una transferencia real de energía desde la red hacia el VE. Durante la transferencia, intercambian continuamente su estado hasta que se cumpla la condición de terminación o se solicite una interrupción por cualquier motivo. En la finalización, realizan controles de seguridad antes de la desconexión.

Durante el procedimiento, el SAVE informa al servidor sobre la sesión de carga. Consulte la Figura 10.11 para ver la secuencia detallada de mensajes durante las fases de configuración, carga y finalización.

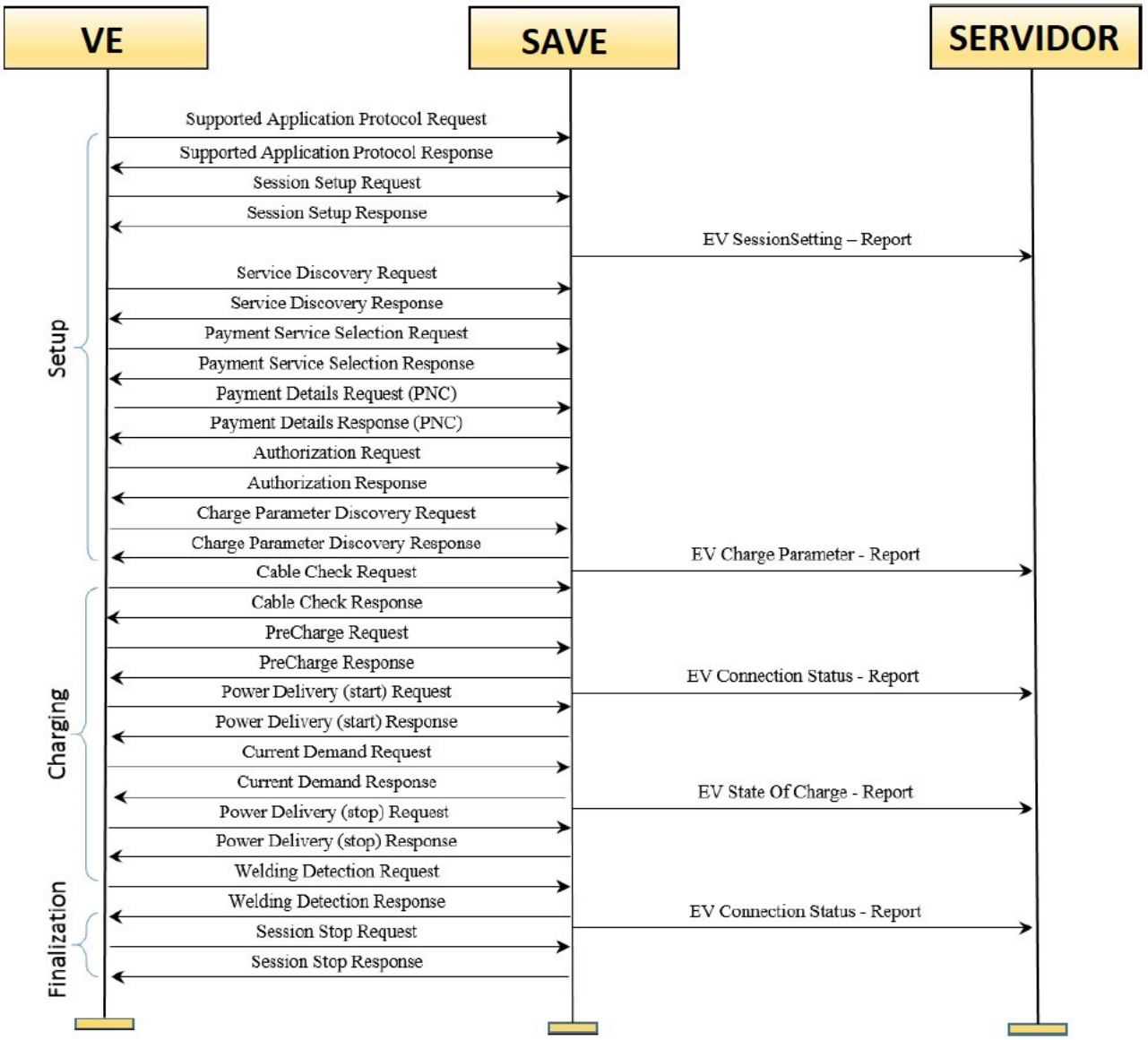


Figura 10.11: Secuencia de mensajes para carga de VE según ISO/IEC 15118.

CAPÍTULO 11

CONCLUSIONES.

11.1. Funcionalidad del SAVE.

El principal propósito del presente proyecto fue desarrollar un cargador para vehículos eléctricos (SAVE) que posibilite la comunicación con el vehículo y con el centro de control de carga. Para ello se debió diseñar e implementar el circuito de comunicación entre el vehículo y el SAVE, así como desarrollar la comunicación entre éste y el centro de control de carga. El cargador implementado permite la carga de un vehículo a tres niveles de potencia, los mismos dependen del conector y de la sección del cable que se utilice para la misma. La carga puede realizarse de forma trifásica a 400 V con neutro o monofásica de 220 V a niveles de corriente máxima de 13 A, 20 A o 32 A. Respecto a la comunicación con el centro de control se resolvió utilizando el lenguaje NodeJS (basado en JavaScript), el cual permitió implementar la misma mediante WebSocket y enviar mensajes con formato JSON de acuerdo a lo planteado por el protocolo OCPP 1.6. Para iniciar una sección de carga se requiere que el usuario cuente con una tarjeta acreditada por el centro de control de carga lo que posibilita el control de utilización del SAVE dependiendo de las condiciones o restricciones que a este se le aplique. Como ejemplo se puede mencionar que si una flota de vehículos eléctricos cuenta con su propio parque de cargadores y centro de control de carga, se podría limitar el uso a determinados vehículos dependiendo del horario y de la demanda de energía del servicio, entre otras condiciones que se consideren pertinente.

11.2. Tiempo de Fabricación y Costos.

El montaje del SAVE puede dividirse en tres etapas, una de estas es el montaje del circuito de potencia que implica la conexión de las llaves de protección, el contactor, los transformadores de corriente,el medidor de energía y el conector de salida. El tiempo estima para este montaje es de aproximadamente tres horas dependiendo de las destreza de quien lo realice. Por otra parte está el armado del circuito de comunicación que implica soldar los componentes de circuito en una placa perforada o de circuito impreso. El tiempo dependerá de cual de los opciones se implemente ya que se estima un menor tiempo en el segundo caso. Además esta sujeto a las habilidades de quien lo realice. En esta oportunidad se utilizó una placa perforada y el tiempo

insumido para realizarlo fue de dos horas y media aproximadamente. Además del montaje del circuito IEC hay que armar la placa que se usa para las conexiones al Arduino y los cables con los conectores lo que insumió un tiempo aproximado de seis horas.

Finalmente el armado de todo el sistema de control que implica la conexión de las fuentes de alimentación, el lector de tarjeta, la pantalla, el regulador de tensión para la alimentación del Arduino y el relé para energizar el contactor se realizo en un tiempo estimado de dos horas.

Los tiempos estimados para la realización de las tres etapas de montaje mencionado suma un total de trece horas aproximadamente. Por lo que se puede decir que el SAVE desarrollado es de fácil montaje. Es claro que en los tiempos presentados previamente no se está teniendo en cuenta el tiempo requerido para el desarrollo de ninguna de las etapas de construcción y no contempla modificaciones de software o hardware.

Los costos de los componentes tiene una variación considerable dependiendo de la marca y calidad de los mismos. En el presupuesto realizado inicialmente se consideraron componente de alta calidad y se estimo un costo de cincuenta y cinco mil pesos. Por razones de presupuesto y tiempo se compraron y utilizaron componente de calidad inferior reduciendo el valor anterior a veinticinco mil pesos.

Es importante destacar que todos los componentes utilizados se pueden obtener en el mercado local lo que hace que sea más sencillo el procesos de armado del SAVE.

11.3. SAVE Desarrollado vs SAVE Comercial.

En esta sección se pretende plantear algunos puntos comparativos entre el cargador desarrollado y uno comercial que está homologado en Uruguay. Sin entrar en detalles de marca y modelos se realizará dicha comparación en términos generales.

Comenzando por los puntos en común se puede destacar el cumplimiento con el protocolo IEC para la comunicación con el vehículo. Así también cumple con las funciones básicas de comunicación requerida por el protocolo OCPP 1.6 para la comunicación con el centro de control de carga. Respecto a este punto, a diferencia de los cargadores comerciales, el SAVE desarrollado no implementa la utilización de la lista blanca y el caché previstas por el mencionado protocolo para su funcionamiento en caso de perder la comunicación con el servidor del centro de control de carga. Lo cual en posteriores versiones puede ser incorporadas sin mayor dificultad.

En término de potencia máxima que puede suministrar, el SAVE desarrollado se encuentra en un punto medio respecto a las opciones que actualmente se disponen en Uruguay, en este caso es trifásica 22 kW.

Respecto al control y la comunicación con el servidor del centro de control de carga se puede encontrar, en los cargadores comerciales, básicamente dos modalidades de implementación. Una de esta utiliza PLCs para implantar el control, la comunicación con el vehículo y la comunicación con el centro de control de caga. Mientras que la otra opción utiliza bloques de circuitos embebidos desarrollados específicamente para implementar cada una de las funcionalidades del cargador. El SAVE desarrollado en esta oportunidad se encuentra más próximo a este último formato de implementación, si bien solo se desarrolló parte del circuito de comunicación con el vehículo, siendo los otros componente de control y comunicación un Arduino y una Raspberry PI. Se optó por seguir este camino por considerase de menor costo que la implementación mediante PLC.

En términos generales se considera que se logró desarrollar un dispositivo confiable con prestaciones similares a las ofrecidas por otros cargadores en el mercado, de bajo costo y que

posibilita una mayor versatilidad en los requerimientos del usuario ya que puede ser adaptado a las necesidades de éste. Sin dudas que queda un largo camino por recorrer para agregarle mejoras y funcionalidades tanto a nivel de software como de hardware. Entre ellas se puede mencionar agregar las funcionalidades requeridas para el funcionamiento sin comunicación con el centro de control de carga, que los circuito de comunicación sean desarrollados específicamente para el propósito requerido, también se puede aumentar la potencia máxima que puede suministrar, entre otras.

11.4. Posibles Mejoras a Futuro.

En esta sección se presentan otras mejoras que se podrían aplicar al SAVE para mejorar su rendimiento, su estética y su robustez entre otras características.

Para poder mejorar las dimensiones del SAVE y lograr una estructura más pequeña se podría cambiar el medidor y los transformadores de corriente por otros con similares características pero de menor tamaño. En este caso no fue posible utilizar esos modelos debido a que se priorizó bajar los costos del SAVE.

En términos de software, el próximo paso imprescindible a implementar sería el de generar una base de datos propia del SAVE donde se pueda almacenar una lista de usuarios validados por el centro de control de carga para lograr autonomía de funcionamiento al momento de no contar con comunicación con el servidor, es decir, un funcionamiento "Offline". Para ello el SAVE debe ser capaz de registrar cada uno de los mensajes que se envían en una comunicación normal con el servidor en todo un proceso de carga y una vez que se retome la comunicación con el mismo se debe enviar toda esta información al servidor reconstruyendo cada una de las sesiones de carga realizadas mientras persistió la desconexión.

En términos del hardware, con la Raspberry PI se podría tratar de implementar todas las comunicaciones y evitar usar el Arduino, con el objetivo de reducir posibles fallas.

La Raspberry Pi3 tiene 40 pines GPIO (entrada-salida de propósito general) que soportan los protocolos SPI (utilizado para lectura de tarjeta), I2C (utilizado para controlar display) y permite generar señales PWM de hasta 3,3 V.

El único inconveniente es que la Raspberry no tiene pines analógicos, son todos digitales, por lo que será necesario algún conversor A/D para leer el voltaje de la parte alta de la señal PWM en el Control Pilot del conector y el voltaje en el Proximity Pilot para medir la resistencia estandarizada, que determina la corriente máxima admisible del conector.

11.5. Competencias Adquiridas.

Respecto a las competencias adquiridas durante el desarrollo del proyecto es posible distinguir entre generales y específicas. Entre las competencias generales adquiridas se puede destacar el fortalecimiento del trabajo en grupo, la planificación y ejecución de un proyecto partiendo de la formulación de los objetivos, la delimitación del alcance, el análisis de posibles obstáculos y/o contratiempos, evaluación de los recursos requeridos y disponibles, planificación temporal, etc.

En términos específicos se considera que el perfil óptimo de los integrantes del grupo debió incluir uno que estuviera más orientado a la programación, otro a la electrónica y otro a potencia. Dado que los tres integrantes del grupo tenemos un perfil en potencia fue necesario entrar en

conocimiento de diversas herramientas requeridas para el desarrollo del cargador, entre ellas se podría mencionar la programación en JavaScript, protocolos de comunicación entre dispositivos (como MODBUS RS-485), programación para Arduino, configuración de dispositivos de red, entre otras. Salvado las distancias se considera que el cúmulo de conocimiento, estrategias de aprendizaje y resolución de problemas adquiridos en el transcurso de la formación de grado fue de gran aporte a la hora de abordar los desafíos que fueron surgiendo durante el proyecto.

11.6. Sobre la Bidireccionalidad.

11.6.1. Condiciones en el Mercado Uruguayo.

Partiendo de la base de que la UNIT adoptó la normativa europea IEC 61851 y establece el conector Tipo 2 o Mennekes (IEC 62196-2) como formato normalizado. Y sabiendo que recién en 2025 (según CharIN) se plantea que los conectores CCS soporten la segunda versión de ISO 18115; entonces en Uruguay, para poder implementar la funcionalidad V2G con los protocolos antes mencionados y con los conectores normalizados, habrá que esperar todavía.

Es posible lograr una descarga de baterías de VE, mediante otro protocolo, con los conectores CHAdeMo que cargan y descargan en DC. Pero no cumple el objetivo de implementar los cambios en el SAVE ya diseñado, porque el SAVE tiene otro tipo de conector y un protocolo comunicación diferente.

Por otro lado, existe el impedimento de que el servidor de UTE no tiene implementado el protocolo OCPP 2.0 que es el que define los mensajes V2G entre el SAVE y el servidor del gestor de carga (UTE). Por lo que dicha comunicación, tampoco es viable implementar aún.

Lo que ya se podría implementar, en el SAVE ya diseñado, es la comunicación definida en la primera versión del protocolo ISO 15118. En caso de implementarse, una mejora posible es que el usuario no tendrá necesidad de pasar una tarjeta para iniciar o detener la sesión. Y dicha funcionalidad (denominada "**Plug&Charge**"), brinda mayor seguridad al sistema frente a hackeos, porque en cada sesión, hay una transacción de credenciales entre el VE y el SAVE.

Además, al implementar esta funcionalidad, se conocerá la comunicación en base a PLC, que será necesaria para implementar la funcionalidad V2G, cuando se publique la segunda versión del protocolo mencionado (en la segunda mitad del 2021).

¿Cuál es la Norma que Regula la Forma de Onda y Armónicos?

A la descarga de baterías de VE, se lo debería considerar como Microgeneración y regirse en el marco del reglameto del 2018 Instalaciones de Microgeneración conectadas a la red de Baja Tensión de UTE - Capítulo XXVIII ", (ver [44]) pero la misma establece que:

- La presente reglamentación se refiere a los requisitos técnicos que deberán cumplir las Instalaciones de Microgeneración, en instalaciones interiores de suscritores existentes, para su conexión a la Red de Distribución de Baja Tensión perteneciente a UTE.

  Es complementaria a los Requisitos Generales fijados por el Ministerio de Industria, Energía y Minería, en el marco del Decreto 173/010.

- Instalación de Microgeneración (IMG): Instalación que dispone de un equipamiento que convierte energía de Fuentes Renovables en energía eléctrica, para cumplir con las condiciones establecidas en el Decreto 173/010.

- Fuentes Renovables: Fuentes de generación provenientes de recursos eólico, solar, biomasa o mini- hidráulica.

El artículo primero de dicho decreto (ver [45]) establece:

"Se autoriza a los suscritores conectados a la red de distribución de baja tensión a instalar generación de origen renovable eólica, solar, biomasa o mini hidráulica. La corriente máxima de régimen generada en baja tensión por los equipos instalados no deberá superar los 16 amperios, con excepción de los suministros monofásicos en redes con la configuración de retorno por tierra, en los que la corriente máxima de régimen será 25 amperios. Asimismo, la potencia pico del equipamiento de generación instalado deberá ser menor o igual a la potencia contratada por el suscritor."

Asimismo, en el reglameto de UTE se establece que los límites de tensiones nominales de utilización en las Instalaciones de Microgeneración son:

- Corriente alterna: Igual o inferior a 1000 V.
- Corriente continua: Igual o inferior a 1500 V.

Y tiene establecido "requerimientos de calidad de onda [1] para las Unidades Generadoras", que complementan lo establecido en el decreto:

Corriente Asignada de Unidad Generadora	Flicker	Armónicos de corriente
Hasta 16A	IEC 61000-3-3	IEC 61000-3-2
Mayor a 16 A, hasta 75 A	IEC 61000-3-11	IEC 61000-3-12
Mayor a 75 A	IEC 61000-3-5	IEC 61000-3-4

Tabla 11.1: Requerimientos de Calidad de onda en Microgeneración.

[1] Observar que el protocolo IEC 61000 es el que los diseños de OBC deben cumplir (ver sección 10.5.2)

11.6.2. Apuntes Finales.

Una posible modificación al SAVE es comenzar a implementar la primera versión del protocolo ISO 18115 para conocer la tecnología basada en comunicación de alto nivel en PLC, entre el VE y el SAVE. Implicaría tanto cambio de hardware, para implementar la señal PLC; como también nueva implementación de software, para decodificar los mensajes recibidos y poder codificar los mensajes para enviar.

Otra modificación posible es implementar el Protocolo OCPP 2.0 en el SAVE, porque dicho protocolo será compatible con la segunda versión del protocolo ISO/IEC 15118. En dicho caso, el inconveniente será que UTE tiene que realizar una actualización de dicho protocolo también. Actualmente el protocolo de comunicación entre el SAVE y UTE es OCPP 1.6. Se permite una gestión inteligente de la carga del VE, pero no esta prevista la descarga. Si UTE no lo actualiza el protocolo OCPP, se pueden enviar los mensajes que el OCPP 1.6 no soporta, en el mensaje DataTransfer.json y pre-establecer el formato para que el SAVE y UTE se puedan comprender y UTE pueda responder a la solicitudes del SAVE.

Protocolo de Comunicación Entre el SAVE y el VE, en el Mercado Local.

Por otro lado, es posible que en el mercado local, se aprueben los estándares europeos y por lo tanto, se apruebe el protocolo ISO/IEC 15118. Si eso sucede, es razonable que se aprueben los conectores macho y hembra CCS combo-2; que habilitarían las cargas ultra-rápidas en DC, con el protocolo IEC 61851-3.

¿Regulación Específica para V2G?

Recientemente fue aprobado el decreto N° 025_2020 y el reglamento de UTE R20-258 (los cuales se pueden ver en el anexo) que entre otras cosas autoriza a los suscriptores conectados a la red de distribución de baja tensión a generar energía eléctrica a partir de una instalación de de baterías que opere en paralelo y que no inyecten energía a la red de distribución.

Anexos

ANEXO A

A.1. Descripción del Arduino.

En esta sección, se hará una breve descripción de la placa Arduino UNO (ver [9] [10] [11]):

¿Que es Arduino?

La placa Arduino es una plataforma de prototipos electrónica de código abierto (open – source) basada en hardware y software flexibles y fáciles de usar. Está pensado e inspirado en artistas, diseñadores, y estudiantes de computación o robótica y para cualquier interesado en crear objetos o entornos interactivo, o simplemente por hobby. Arduino consta de una placa principal de componentes eléctricos, donde se encuentran conectados los controladores principales que gestionan los demás complementos y circuitos ensamblados en la misma. Además, requiere de un lenguaje de programación para poder ser utilizado, programado y configurado, por lo que se puede decir que Arduino es una herramienta "completa", ya que sólo se debe instalar y configurar con el lenguaje de programación de esta placa los componentes eléctricos que se requieran para realizar el proyecto que se tiene en mente, haciéndola una herramienta no sólo de creación, sino también de aprendizaje en el ámbito del diseño de sistemas electrónicos-automáticos y, además, fácil de utilizar. Arduino también simplifica el proceso de trabajo con micro controladores, ya que está fabricada de tal manera que viene "pre ensamblada" y lista con los controladores necesarios para poder operar con ella de forma inmediata, lo que ofrece una ventaja muy grande para profesores, estudiantes y aficionados interesados en el desarrollo de tecnologías. Las posibilidades de realizar proyectos basados en esta plataforma tienen como limite la imaginación de quien opera esta herramienta.

La placa Arduino UNO esta basada en un chip Atmel ATmega328. Entre sus características se encuentran 14 pines digitales de entrada/salida, de los cuales 6 pueden ser utilizadas como salidas PWM (característica utilizada en este proyecto), 6 entradas analógicas, conexión USB, un conector de alimentación (de $9V$), etc.

En la figura A.1 se puede observar la placa de Arduino UNO utilizada.

Figura A.1: Placa Arduino UNO.

A continuación se realizará una breve descripción de la figura A.1 con la ayuda de la numeración que aparece en la misma.

1.“Boton de Reset”: Este botón sirve para inicializar nuevamente el programa cargado en la placa. Cuando la placa no responde correctamente sirve como botón de encendido y apagado.

2 y 3. “Pines o Puertos de Entrada y Salida”: Estos son los pines donde se conectan los sensores, componentes y actuadores que necesiten señales digitales. Como se puede ver en la imagen A.1 hay pines a los cuales los ante sede el símbolo ($\sim$), esto se debe a que esos pines son los que permiten trabajar con señales PWM .

4. “Puerto USB”: Es el puerto que se utiliza para conectar con una computadora y poder transferir o cargar los programas al microcontrolador. También se utiliza para darle energía a la placa y para la transferencia serie de la misma, tanto para trasmisión como recepción de datos.

5.“Chip de Interfase USB”: Este chip es el que se encarga de controlar la comunicación del puerto USB.

6. “Reloj Oscilador”: Es el dispositivo encargado de marcar el ritmo al cual se deben ir ejecutando las instrucciones.

7. “Led de Encendido”: Es un pequeño led que al encenderse da la pauta que la placa esta correctamente alimentada.

8. “Microcontrolador”: Es conocido como el “cerebro”de la placa Arduino. Es el procesador que se encarga de ejecutar las instrucciones del los programas o del programa.

9. “Regulador de Tensión”: Es el dispositivo que regula la tensión en 5 V que se envía a los pines de la placa para asegurar proteger el microcontrolador.

10. "Puerto de Corriente Continua": Es el puerto que se utiliza cuando se necesita energizar la placa a una potencia mayor que la suministrada por el puerto USB.

11. "Zócalo de Tensión": Como se puede ver en la figura A.1 de estos pines se puede obtener un voltaje de 3,3 V (con un consumo de corriente máxima de 50 mA) o un voltaje de 5 V los cuales (en general) se utilizan para energizar el circuito con el que se va a trabajar. También puede verse tres pines con la etiqueta GND la cual hace referencia al pin de tierra. El pin con la etiqueta V_{in} se utiliza como otra opción para alimentar la placa Arduino, la cual debe ser una tensión de 5 V.

12. "Entradas Analógicas": Estos 6 pines son los que le permiten a la placa leer señales del tipo analógico.

Otra característica importante a resaltar de la placa Arduino es la comunicación. Con esta placa se tiene la posibilidad de conectarse con otro Arduino, con una computadora u otros microcontroladores (por ejemplo una Raspberry) de forma muy sencilla.

Una de las formas de comunicación es a través de los pines 0 (RX) y 1 (TX) los que proporcionan una comunicación serie UART TTL (5 V). Las conexiones de estos pines se deben cruzar para conectar con otros dispositivos, es decir, el pin TX del dispositivo 1 se debe conectar con el pin RX del dispositivo 2, por lo tanto, el TX del dispositivo 2 debe de conectarse con el RX del dispositivo 1. Además ambos dispositivos deben compartir una masa común. Se pueden encontrar dispositivos que solo cuenten con el pin TX, eso se debe a que estos dispositivos solo envían datos sin la necesidad de recibir.

Otra manera de comunicación con la placa Arduino es a través de la comunicación serie por el puerto USB. Un ATmega16U2 en la placa canaliza esta comunicación serie a través del USB el cual aparece como un puerto de comunicación virtual con el software del ordenador. El Firmware 16U2 utiliza los controladores COM USB estándar lo que hace que no se necesite ningún controlador externo para su correcta comunicación.

Otro elemento vital que se debe conocer sobre la placa Arduino es el softwar que maneja. La placa Arduino maneja un lenguaje de programación que sirve para controlar los distintos sensores que se encuentran conectados a la placa, por medio de instrucciones y parámetros que previamente el usuario establece creando un programa y subiéndolo a la placa mientras esta conectada a la computadora. El lenguaje con el cual se realiza el programa que opera dentro del Arduino se llama Wiring, el cual esta basado en la plataforma Processing y muy fuertemente basado en el lenguaje de programación C/C++. Cabe destacar que aunque el entorno de desarrollo integrado del Arduino (Arduino IDE) no es la única que nos permite interactuar con la placa Arduino éste es un software gratuito el cual se puede descargar de la misma pagina web de Arduino.

La figura A.2 muestra como se presenta el entorno de desarrollo Arduino IDE al comenzar un nuevo proyecto.

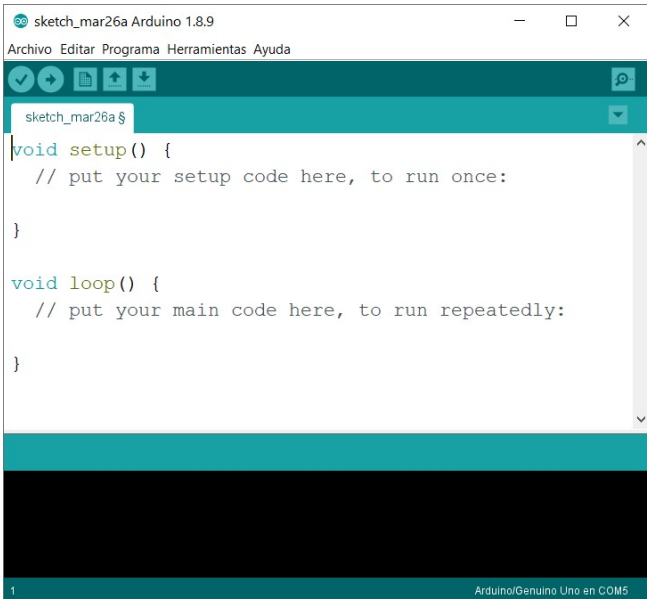


Figura A.2: Entorno de desarrollo Arduino IDE.

Como se puede ver en la figura A.2 al abrir el programa se puede diferencias dos funciones de estructura, la función setup y la función loop, que se explicarán brevemente a continuación.

función setup(): Esta función se ejecuta solo una vez al iniciar el programa. Es en esta función donde se deben inicializar las variables, abrir los canales de comunicación (puerto seria, etc). Es donde se debe cargar el estado inicial del proyecto.

función loop(): Esta función tiene la particularidad que se ejecuta infinitamente. Es decir, que todo lo que este dentro de esta función, el Arduino lo ejecuta una y otra vez. Lo que esté en ésta función se lo denomina cuerpo principal del programa.

A.2. Estructura de Mensajes Según OCPP 1.6.

En esta sección se presenta la lista de todos los mensajes definidos en OCPP 1.6. Además se realizara un breve análisis de algunos de estos mensajes que no han sido implementados por el SAVE diseñado en este proyecto.

- Authorize
- BootNotification
- CancelReservation
- ChangeAvailability
- ChangeConfiguration
- ClearCache
- ClearChargingProfile
- DataTransfer
- DiagnosticsStatusNotification
- FirmwareStatusNotification
- GetCompositeSchedule
- GetConfiguration
- GetDiagnostics
- GetLocalListVersion
- Heartbeat
- MeterValues
- RemoteStartTransaction
- RemoteStopTransaction
- ReserveNow
- Reset
- SendLocalList
- SetChargingProfile
- StartTransaction
- StatusNotification
- StopTransaction
- TriggerMessage
- UnlockConnector
- UpdateFirmware

A.3. Descripción de los Mensajes de Comunicación Según OCCP 1.6.

En esta sección se detallarán todos los mensajes que intervienen en la comunicación entre el SAVE y el centro de control de carga. Cabe resaltar que el protocolo OCCP 1.6 por el cual se rige la comunicación presenta una gran cantidad de mensajes posibles para realizarla. A continuación se presentarán los mensajes implementados en este caso por el SAVE

- Authorize.
- BootNotification.
- Heartbeat.
- StartTransaction.
- MeterValues.
- StopTransaction.

Para poder realizar un correcto análisis y descripción de los mensajes se comenzara analizando la estructura presentada para cada mensaje por el protocolo OCPP 1.6 en donde se verá que dentro de esa estructura se presentan campos que deben ser incluidos de forma obligatoria dentro del mensaje tanto para el que se emite (request) como para el que se recibe (response) y campos que no lo son. Como en este caso, el servidor de UTE implementa la comunicación utilizando solo los campos que son obligatorios, entonces los mensajes desarrollados en el SAVE, solo enviarán mensajes con los campos obligatorios sin agregar otros campos ya que no serían leídos por el servidor y por ende no recibiría una respuesta.

Cabe aclarar que en la siguiente presentación de los mensajes se mostrará la estructura según OCCP 1.6 y como se envía la misma en formato JSON luego de depurar cuales son los campos obligatorios. Luego de presentar cada mensaje se muestra un ejemplo del mensaje a enviar dejando sin completar lo campos con información que depende directamente de la carga que se esta llevando a cabo. La información de estos campos se verá en el ejemplo de comunicación en un proceso normal de carga en la sección 3.4.

A.3.1. Mensaje Authorize Request.

Este mensaje es enviado al momento de querer identificar al usuario. Una vez que el SAVE está listo para iniciar una sesión de carga se le solicita al usuario que pase su tarjeta por el lector. Lo que se muestra a continuación es la estructura del mensaje Authorize request presentada por el protocolo OCCP 1.6.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "AuthorizeRequest",
    "type": "object",
    "properties": {
        "idTag": {
```

```
            "type": "string",
            "maxLength": 20
        }
    },
    "additionalProperties": false,
    "required": [
        "idTag"
    ]
}
```

Como se puede ver el único campo requerido en este mensaje es el "idTag". Esto se puede ver en los campos del mensaje que se ven a continuación;

```
    "required": [
        "idTag"
    ]
```

por lo tanto, la forma del mensaje que cumple con la estructura vista en la sección anterior para poder ser enviado al servidor a través de WebSocket es la siguiente.

```
[
    2,
    "UniqueId",
    "Authorize",
    {
      "idTag":"    "
    }
]
```

El campo que se debe completar es el "idTag" el cual contiene la información del cliente que quiere ser autorizado para comenzar una carga. Lo único que debe cumplir ese campo aparece en el siguiente fragmento de la estructura del mensaje presentada por el protocolo.

```
  "idTag": {
            "type": "string",
            "maxLength": 20
        }
```

Como se puede ver el contenido del campo "idTag" es del tipo "string" (cadena de caracteres) con un largo máximo de 20 caracteres. En el sistema utilizado por UTE el idTag corresponde a identificadores de tarjetas que son otorgadas por UTE.

A.3.2. Mensaje Authorize Response.

Ahora se verá como es la estructura de un mensaje de respuesta a un Authorize. A continuación se muestra como se presenta este mensaje según el protocolo mencionado.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "AuthorizeResponse",
    "type": "object",
    "properties": {
        "idTagInfo": {
            "type": "object",
            "properties": {
                "status": {
                    "type": "string",
                    "enum": [
                        "Accepted",
                        "Blocked",
                        "Expired",
                        "Invalid",
                        "ConcurrentTx"
                    ]
                },
                "expiryDate": {
                    "type": "string",
                    "format": "date-time"
                },
                "parentIdTag": {
                    "type": "string",
                    "maxLength": 20
                }
            },
            "required": [
                "status"
            ]
        }
    },
    "additionalProperties": false,
    "required": [
        "idTagInfo"
    ]
}
```

Como se puede ver en la estructura del mensaje, el objeto “idTagInfo” contiene la información de“status”, “expiryDate” y “parentIdTag”. En el siguiente fragmento de la estructura se puede ver que “idTagInfo” aparece como requerido.

```
"required": [
    "idTagInfo"
]
```

A su vez dentro de “idTagInfo” se puede ver que el campo requerido es el de “status”, esto se puede ver en el fragmento;

```
"required": [
            "status"
        ]
```

Por lo tanto el campo “expiryDate” es opcional y contiene la información de hasta qué fecha el usuario correspondiente al idTag enviado es un usuario valido. El campo “parentIdTag” contiene un identificador principal pero tampoco es usado ya que es un campo opcional. El único campo requerido por el protocolo es el campo “status” el cual pude contener las siguientes respuestas.

Valor de status	Descripción
Accepted	El usuario es reconocido como válido y es aceptado.
Blocked	El usuario está bloqueado. No se le permite la carga.
Expired	La validez para ese usuario ha expirado. No se le permite la carga.
Invalid	El usuario es desconocido. No se le permite la carga.
ConcurrentTx	El usuario ya tiene una sección de carga abierta. No se le permite la carga.

Tabla A.1: Valores posibles para el campo status.

Teniendo en cuenta los campos que son requeridos de forma obligatoria y llevando el mensaje a la estructura utilizada para el envió de los mensajes la respuesta que se espera recibir a un Autorize tiene la siguiente forma.

```
[
    3,
    "UniqueId",
    {
    idTagInfo:
        {
        "status":"   "
        }
    }
]
```

La presentación de los mensajes según el protocolo OCPP 1.6 y de que campos son requeridos para la comunicación es similar para todos los mensajes, por lo tanto, para el resto de los mensajes no se mostrará particularmente en que fragmento del mensaje se muestra los campos requeridos.

A.3.3. Mensaje BootNotification Request.

Este mensaje se utiliza para saber en que “estado” está el SAVE, es decir, si está autorizado por el servidor para poder cargar o si está fuera de servicio. Lo primero que debe de hacer el

SAVE al encenderse o luego de haber perdido conexión y reiniciarse, es enviar al servidor un BootNotification para saber cuál es su situación actual. La forma presentada por el protocolo OCPP 1.6 para el mensaje BootNotification se puede ver a continuación.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "BootNotificationRequest",
    "type": "object",
    "properties": {
        "chargePointVendor": {
            "type": "string",
            "maxLength": 20
        },
        "chargePointModel": {
            "type": "string",
            "maxLength": 20
        },
        "chargePointSerialNumber": {
            "type": "string",
            "maxLength": 25
        },
        "chargeBoxSerialNumber": {
            "type": "string",
            "maxLength": 25
        },
        "firmwareVersion": {
            "type": "string",
            "maxLength": 50
        },
        "iccid": {
            "type": "string",
            "maxLength": 20
        },
        "imsi": {
            "type": "string",
            "maxLength": 20
        },
        "meterType": {
            "type": "string",
            "maxLength": 25
        },
        "meterSerialNumber": {
            "type": "string",
            "maxLength": 25
        }
    },
    "additionalProperties": false,
```

```
    "required": [
        "chargePointVendor",
        "chargePointModel"
    ]
}
```

Analizando la estructura de forma similar a la del mensaje anterior se puede ver en los únicos campos obligatorios que se deben enviar en el mensaje son “chargePointModel” y “chargePointVendor”, más allá de que se podrían enviar otros campos opcionales como lo son por ejemplo el “chargePointSerialNumber”, “firmwareVersion” , etc. Usando los campos obligatorios y el formato adecuado para enviar el mensaje se llega a la siguiente estructura de mensaje JSON.

```
[
    2,
    "UniqueId",
    "BootNotification",
    {
      "chargePointModel":"     ",
      "chargePointVendor":"     "
    }
]
```

Los campos“chargePointModel” y “chargePointVendor” contienen información del SAVE la cual el servidor chequea en su base de datos para determinar si es un cargador que este autorizado dentro de su sistema. Recordar que los ejemplos que se están presentando no contienen ninguna información particular de cada SAVE, el contenido de los campos “chargePointModel” y “chargePointVendor” se pueden ver en la sección 3.4 para el caso del SAVE desarrollado en este proyecto.

A.3.4. Mensaje BootNotification Response.

La respuesta al BootNotification es muy importante, ya que le da la información al cargador de su estado. La estructura completa de un BootNotification response se puede ver a continuación.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "BootNotificationResponse",
    "type": "object",
    "properties": {
        "status": {
            "type": "string",
            "enum": [
                "Accepted",
                "Pending",
```

```
                "Rejected"
            ]
        },
        "currentTime": {
            "type": "string",
            "format": "date-time"
        },
        "interval": {
            "type": "number"
        }
    },
    "additionalProperties": false,
    "required": [
        "status",
        "currentTime",
        "interval"
    ]
}
```

En este caso los campos “status”, “currentTime” e “interval” son todos de requerimiento obligatorio, es decir que deben aparecer dentro del mensaje de respuesta. El campo “current-Time” se utiliza para enviar la fecha y la hora correspondiente por parte del servidor para que el SAVE (si lo desea) pueda usar como referencia.

El campo correspondiente a “interval” se utiliza para enviar un valor de tiempo, por ejemplo, 5 minutos. Este valor se utiliza para saber cada cuanto tiempo el SAVE debe enviarle una señal de actividad al servidor. En el caso que el SAVE luego de enviar un BootNotification es aceptado el SAVE debe enviar al servidor el mensaje HeartBeat cada intervalos de tiempo “interval”, en este ejemplo se debería de enviar ese mensaje cada 5 minutos. Este chequeo llamado “latido de corazón” se utiliza para saber si la comunicación sigue funcionando correctamente.

En el caso que el SAVE no sea aceptado luego de mandar un BootNotification el mensaje que debe enviar cada intervalos de tiempo “Interval” es nuevamente el BootNotification. En este ejemplo si por alguna razón que se verá más adelante la respuesta al BootNotification no fue la de aceptado el SAVE debe cada 5 minutos enviar un BootNotification al servidor para intentar ser aceptado.

El ultimo campo que queda por analizar en la respuesta del mensaje BootNotification es el campo “status”. Los posibles valores para este campo se muestran en la tabla.

Valor de status	Descripción
Accepted	El SAVE es aceptado por el servidor.
Pending	El servidor todavía no esta listo para aceptar al SAVE. El servidor puede aceptar al SAVE en algún momento.
Rejected	El SAVE ha sido rechazado. Puede ser que sea un SAVE desconocido para el servidor.

Tabla A.2: Valores posibles para el campo status del mensaje BootNotification.

Por lo tanto, la forma del mensaje JSON para la respuesta a un BootNotification será la siguiente.

```
[
    3,
    "UniqueId",
    {
        "status":" ",
        "currentTime":" ",
        "interval": ,
    }
]
```

Recordar que la información contenida en los campos “status”, “currentTime” e “interval” es particular para cada carga. Un ejemplo se va a ver en la sección 3.4.

A.3.5. Mensaje HeartBeat Request.

Este mensaje se utiliza como método de control por parte del servidor para chequear que el SAVE continua con una comunicación activa. Una vez que el servidor le envía al SAVE (a través del BootNotificaton response) un intervalo de tiempo determinado, el SAVE debe enviar el mensaje Heartbeat cada ese período de tiempo. La estructura del mensaje Heartbeat request según el protocolo OCPP 1.6 se presenta a continuación.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "HeartbeatRequest",
    "type": "object",
    "properties": {},
    "additionalProperties": false
}
```

Como se puede ver en la estructura del mensaje correspondiente a un herbeat request no contiene ningún campo donde se envíe algún tipo de información, por lo tanto la forma del mensaje a enviar será la siguiente.

```
[
    2,
    "UniqueId",
    "Heartbeat",
    {
    }
]
```

A.3.6. Mensaje HeartBeat Response.

Este mensaje es enviado por parte del servidor como respuesta al Heartbeat enviado por el SAVE. La estructura del mensaje Heartbeat response según el protocolo se muestra a continuación.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "HeartbeatResponse",
    "type": "object",
    "properties": {
        "currentTime": {
            "type": "string",
            "format": "date-time"
        }
    },
    "additionalProperties": false,
    "required": [
        "currentTime"
    ]
}
```

Como se puede ver el mensaje de respuesta por parte del servidor a un Heartbeat request solo contiene como información la fecha y hora, por lo tanto la forma del mensaje JSON que recibirá el SAVE por parte del servidor es la siguiente.

```
[
    3,
    "UniqueId",
    {
    "currentTime":" "
    }
]
```

A.3.7. Mensaje StartTransaction Request

Este mensaje se envía por parte del SAVE al momento de comenzar una transacción. El mensaje StartTransaction se envía luego de que un usuario es validado (por medio de un mensaje Authorize) y que se haya constatado la correcta conexión de un VE con un conector que cumpla con las condiciones de carga del SAVE. Una vez verificadas estas cosas el SAVE envía un StartTransaction cuya estructura completa se puede ver a continuación.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "StartTransactionRequest",
    "type": "object",
    "properties": {
        "connectorId": {
            "type": "integer"
        },
        "idTag": {
            "type": "string",
            "maxLength": 20
        },
        "meterStart": {
            "type": "integer"
        },
        "reservationId": {
            "type": "integer"
        },
        "timestamp": {
            "type": "string",
            "format": "date-time"
        }
    },
    "additionalProperties": false,
    "required": [
        "connectorId",
        "idTag",
        "meterStart",
        "timestamp"
    ]
}
```

Como se puede ver en la estructura del mensaje los campos obligatorios requeridos en el mensaje son “connectorId”, “IdTag”, “meterStart” y “timestamp”. Los campos “connectorId”, “IdTag” y “timestamp” ya se vieron en mensajes anteriores, el campo “meterStart” se utiliza para enviar el valor de energía acumulado en el medidor para ser tomado como punto de partida al momento de iniciar una nueva carga por parte del SAVE. La forma del mensaje JSON para un StartTransaction request se muestra a continuación.

```
[
```

```
  2,
  "UniqueId",
  "StartTransaction",
  {
    "connectorId": ,
    "idTag":" ",
    "meterStart": ,
    "timestamp":" "
  }
]
```

A.3.8. Mensaje StartTransaction Response.

Este mensaje determina si la transacción va a poder ser iniciada o no. La estructura completa del mensaje es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "StartTransactionResponse",
    "type": "object",
    "properties": {
        "idTagInfo": {
            "type": "object",
            "properties": {
                "expiryDate": {
                    "type": "string",
                    "format": "date-time"
                },
                "parentIdTag": {
                    "type": "string",
                    "maxLength": 20
                },
                "status": {
                    "type": "string",
                    "enum": [
                        "Accepted",
                        "Blocked",
                        "Expired",
                        "Invalid",
                        "ConcurrentTx"
                    ]
                }
            },
            "required": [
                "status"
            ]
        },
```

```
"transactionId": {
            "type": "integer"
        }
    },
    "additionalProperties": false,
    "required": [
        "idTagInfo",
        "transactionId"
    ]
}
```

Los campos que aparecen en este caso como requeridos son “idTagInfo” y “transactionId”. El campo “idTagInfo” es similar al visto en el mensaje Authorize su valor corresponde a los de la tabla A.1. El campo “transactionId” contiene un valor asignado por parte del servidor como un Id de la transacción. Este valor va a ser utilizado posteriormente para poder “cerrar” la transacción.

A continuación se muestra la estructura principal del mensaje JSON recibido por el SAVE por parte del servidor.

```
[
    3,
    "UniqueId",
    "transactionId": ,
    {
        "idTagInfo",
        {
        "status":" "
        }
    }
]
```

A.3.9. Mensaje MeterValues Request.

El mensaje MeterValue se utiliza para enviar al servidor la información de la energía, corriente, potencia, etc, que el VE va consumiendo en el transcurso de la carga. A continuación se presenta la estructura del mensaje según el protocolo OCPP 1.6.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "MeterValuesRequest",
    "type": "object",
    "properties": {
        "connectorId": {
            "type": "integer"
        },
        "transactionId": {
            "type": "integer"
```

```
},
"meterValue": {
    "type": "array",
    "items": {
        "type": "object",
        "properties": {
            "timestamp": {
                "type": "string",
                "format": "date-time"
            },
                "sampledValue": {
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "value": {
                            "type": "string"
                        },
                        "context": {
                            "type": "string",
                            "enum": [
                                "Interruption.Begin",
                                "Interruption.End",
                                "Sample.Clock",
                                "Sample.Periodic",
                                "Transaction.Begin",
                                "Transaction.End",
                                "Trigger",
                                "Other"
                            ]
                        },
                        "format": {
                            "type": "string",
                            "enum": [
                                "Raw",
                                "SignedData"
                            ]
                        },
                        "measurand": {
                            "type": "string",
                                "enum": [
                                "Energy.Active.Export.Register",
                                "Energy.Active.Import.Register",
                                "Energy.Reactive.Export.Register",
                                "Energy.Reactive.Import.Register",
                                "Energy.Active.Export.Interval",
```

```
                "Energy.Active.Import.Interval",
                "Energy.Reactive.Export.Interval",
                "Energy.Reactive.Import.Interval",
                "Power.Active.Export",
                "Power.Active.Import",
                "Power.Offered",
                "Power.Reactive.Export",
                "Power.Reactive.Import",
                "Power.Factor",
                "Current.Import",
                "Current.Export",
                "Current.Offered",
                "Voltage",
                "Frequency",
                "Temperature",
                "SoC",
                "RPM"
            ]
        },
        "phase": {
            "type": "string",
                "enum": [
                "L1",
                "L2",
                "L3",
                "N",
                "L1-N",
                "L2-N",
                "L3-N",
                "L1-L2",
                "L2-L3",
                "L3-L1"
            ]
        },
        "location": {
            "type": "string",
            "enum": [
                "Cable",
                "EV",
                "Inlet",
                "Outlet",
                "Body"
            ]
        },
        "unit": {
            "type": "string",
```

```
                                    "enum": [
                                        "Wh",
                                        "kWh",
                                        "varh",
                                        "kvarh",
                                        "W",
                                        "kW",
                                        "VA",
                                        "kVA",
                                        "var",
                                        "kvar",
                                        "A",
                                        "V",
                                        "K",
                                        "Celcius",
                                        "Fahrenheit",
                                        "Percent"
                                    ]
                                }
                            },
                            "required": [
                                "value"
                            ]
                        }
                    }
                },
                "required": [
                    "timestamp",
                    "sampledValue"
                ]
            }
        }
    },
    "additionalProperties": false,
    "required": [
        "connectorId",
        "meterValue"
    ]
}
```

En este caso que la estructura del mensaje es bastante amplia se realizará el análisis por parte de forma similar a como se hizo para el primer mensaje. Como se puede ver al final de la estructura, aparecen como requeridos los siguientes campos:

```
"required": [
     "connectorId",
     "meterValue"
```

```
    ]
```

A su vez, el campo “meterValue” (que es de tipo array) tiene los siguientes campos como requeridos:

```
"required": [
                    "timestamp",
                    "sampledValue"
                ]
```

Dentro del campo “sampledValue” también aparece un campo requerido:

```
 "required": [
              "value"
        ]
```

Por lo tanto, los campos que se deben de enviar en el mensaje son “connectorId”, “meterValuee”, “sampledValue”, “value” y “timestamp”. Cabe resaltar que en este caso también se utilizan los campos “measurand” y “unit” para enviar la información de a que magnitud corresponde el valor enviado y en que unidades esta. En el caso del MeterValue Request enviado por el SAVE el valor que se envía es el de la energía ya que es el dato que el servidor (de UTE, en este caso) espera recibir en ese mensaje. En base a todo lo analizado anteriormente, la forma del mensaje JSON para el MeterValue request es la siguiente.

```
[
    2,
    "UniqueId",
    "MeterValues",
    {
    "connectorId": ,
    "meterValue":
    [
        {
        "sampledValue":
        [
            {
            "measurand":"Energy.Active.Import.Register",
            "unit":"Wh",
            "value":" "
            }
        ],
        "timestamp":" "
        }
    ]
  }
]
```

Como se puede ver el campo “conectorId” se utiliza para saber (cuando el SAVE tiene más de un conector) a cuales de los conectores hace referencia esa medida de energía, ya que podría haber dos VE cargando de forma simultanea en el mismo SAVE. Luego el campo “measurand” se utiliza para saber a que hace referencia el dato que se va a enviar (en este caso energía activa), el campo “unit” lleva la unidad correspondiente al valor medido y el campo “value” se utiliza para enviar el dato el cual es tomado con el medidor de energía.

Los mensajes MeterValues son enviados por parte del SAVE una vez que una sesión de carga es iniciada correctamente y dejan de enviarse al momento de cerrar la sesión de carga. Los mismos se envían cada intervalos de tiempo establecidos por el servidor, en este caso se envían cada un minuto.

A.3.10. Mensaje MeterValues Response.

Este mensaje es enviado por parte del servidor como respuesta al metervalue request. A continuación se muestra la estructura completa del mensaje.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "MeterValuesResponse",
    "type": "object",
    "properties": {},
    "additionalProperties": false
}
```

Como se puede ver es un mensaje sin campos con información adicional, solo se envía un mensaje "vacio" como respuesta. Por lo tanto la forma del mensaje JSON que recibe el SAVE es la siguiente.

```
[
    3,
    "UniqueId",
    {
    }
]
```

Este mensaje es similar al que se recibe como respuesta a un Heartbeat.

A.3.11. Mensaje StopTransaction Request.

Este mensaje se envía por parte del SAVE para detener una transacción, por lo tanto, para enviar este mensaje es imprescindible que una transacción halla sido iniciada. A instancias de una transacción iniciada la misma se puede detener si se desconecta el VE o si el usuario pasa la tarjeta para detener la misma. En estos casos se envía un StopTransaction request.

A continuación se muestra la estructura completa del mismo.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "StopTransactionRequest",
    "type": "object",
    "properties": {
        "idTag": {
            "type": "string",
            "maxLength": 20
        },
        "meterStop": {
            "type": "integer"
        },
        "timestamp": {
```

```
            "type": "string",
            "format": "date-time"
        },
        "transactionId": {
            "type": "integer"
        },
        "reason": {
            "type": "string",
            "enum": [
                "EmergencyStop",
                "EVDisconnected",
                "HardReset",
                "Local",
                "Other",
                "PowerLoss",
                "Reboot",
                "Remote",
                "SoftReset",
                "UnlockCommand",
                "DeAuthorized"
            ]
        },
        "transactionData": {
            "type": "array",
            "items": {
                "type": "object",
                "properties": {
                    "timestamp": {
                        "type": "string",
                        "format": "date-time"
                    },
                    "sampledValue": {
                        "type": "array",
                        "items": {
                            "type": "object",
                            "properties": {
                                "value": {
                                    "type": "string"
                                },
                                "context": {
                                    "type": "string",
                                    "enum": [
                                        "Interruption.Begin",
                                        "Interruption.End",
                                        "Sample.Clock",
                                        "Sample.Periodic",
```

```
                "Transaction.Begin",
                "Transaction.End",
                "Trigger",
                "Other"
            ]
        },
        "format": {
            "type": "string",
            "enum": [
                "Raw",
                "SignedData"
            ]
        },
        "measurand": {
            "type": "string",
            "enum": [
                "Energy.Active.Export.Register",
                "Energy.Active.Import.Register",
                "Energy.Reactive.Export.Register",
                "Energy.Reactive.Import.Register",
                "Energy.Active.Export.Interval",
                "Energy.Active.Import.Interval",
                "Energy.Reactive.Export.Interval",
                "Energy.Reactive.Import.Interval",
                "Power.Active.Export",
                "Power.Active.Import",
                "Power.Offered",
                "Power.Reactive.Export",
                "Power.Reactive.Import",
                "Power.Factor",
                "Current.Import",
                "Current.Export",
                "Current.Offered",
                "Voltage",
                "Frequency",
                "Temperature",
                "SoC",
                "RPM"
            ]
        },
        "phase": {
            "type": "string",
            "enum": [
                "L1",
                "L2",
                "L3",
```

```
                            "N",
                            "L1-N",
                            "L2-N",
                            "L3-N",
                            "L1-L2",
                            "L2-L3",
                            "L3-L1"
                        ]
                    },
                    "location": {
                        "type": "string",
                        "enum": [
                            "Cable",
                            "EV",
                            "Inlet",
                            "Outlet",
                            "Body"
                        ]
                    },
                    "unit": {
                        "type": "string",
                        "enum": [
                            "Wh",
                            "kWh",
                            "varh",
                            "kvarh",
                            "W",
                            "kW",
                            "VA",
                            "kVA",
                            "var",
                            "kvar",
                            "A",
                            "V",
                            "K",
                            "Celcius",
                            "Fahrenheit",
                            "Percent"
                        ]
                    }
                },
                "required": [
                    "value"
                ]
            }
        }
```

```
                },
                "required": [
                    "timestamp",
                    "sampledValue"
                ]
            }
        }
    },
    "additionalProperties": false,
    "required": [
        "transactionId",
        "timestamp",
        "meterStop"
    ]
}
```

Como se puede ver en la estructura del mensaje los campos obligatorios para enviar el StopTransaction son “transactionId”, “timestamp” y “meterStop”. Como ya se vio, el valor del campo “transactionId” viene dado en la respuesta del StartTransaction. Es imprescindible que para detener una transacción estos dos valores sean los mismos ya que es la manera de referenciar que transacción se quiere detener. El campo correspondiente a “meterStop” se utiliza para enviar el ultimo dato de energía tomado por el medidor al momento de enviar el StoptTransaction.

Por lo tanto la estructura elemental de forma general del mensaje JSON que debe enviar el SAVE al servidor es la siguiente.

```
[
  2,
  "UniqueId",
  "StopTransaction",
  {
    "transactionId": ,
    "timestamp":" ",
    "meterStop":
  }
]
```

A.3.12. Mensaje StopTransaction Response.

Este mensaje es enviado por parte del servidor como respuesta al StopTransaction request enviado previamente por el SAVE. La forma para este mensaje según el protocolo OCPP 1.6 es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "StopTransactionResponse",
    "type": "object",
    "properties": {
```

```
        "idTagInfo": {
            "type": "object",
            "properties": {
                "expiryDate": {
                    "type": "string",
                    "format": "date-time"
                },
                "parentIdTag": {
                    "type": "string",
                    "maxLength": 20
                },
                "status": {
                    "type": "string",
                    "enum": [
                        "Accepted",
                        "Blocked",
                        "Expired",
                        "Invalid",
                        "ConcurrentTx"
                    ]
                }
            },
            "required": [
                "status"
            ]
        }
    },
    "additionalProperties": false
}
```

Como se puede ver, el campo requerido en este caso es el campo “status” el cual puede tomar los valores según la tabla A.1 que se presento al hacer referencia al mensaje Authorize. Por lo tanto, la estructura general del mensaje JSON será la siguiente.

```
[
    3,
    "UniqueId",
    {
    "idTagInfo",
        {
        "status":" ",
        }
    }
]
```

Con esto quedan presentados en detalle los mensajes que hacen a la comunicación entre el SAVE y el centro de control de carga. No obstante, como se comento al comienzo, hay una gran cantidad de mensajes presentes en el protocolo los cuales no han sido implementados.

A.3.13. ReserveNow.

Este mensaje ha sido comentado anteriormente. Como se vio, se utiliza para que un usuario pueda reservar el SAVE para el horario que lo necesite. Esto es de gran ayuda para el usuario ya que le permite saber que al llegar al lugar del SAVE va a poder cargar su VE ya que realizó una reserva previa del mismo.

A.3.13.1. ReserveNow Request.

La estructura del mensaje de tipo request es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "ReserveNowRequest",
    "type": "object",
    "properties": {
        "connectorId": {
            "type": "integer"
        },
        "expiryDate": {
            "type": "string",
            "format": "date-time"
        },
        "idTag": {
            "type": "string",
            "maxLength": 20
        },
        "parentIdTag": {
            "type": "string",
            "maxLength": 20
        },
        "reservationId": {
            "type": "integer"
        }
    },
    "additionalProperties": false,
    "required": [
        "connectorId",
        "expiryDate",
        "idTag",
        "reservationId"
    ]
}
```

A.3.13.2. ReserveNow Response.

La estructura del mensaje de tipo response es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "ReserveNowResponse",
    "type": "object",
    "properties": {
        "status": {
            "type": "string",
            "enum": [
                "Accepted",
                "Faulted",
                "Occupied",
                "Rejected",
                "Unavailable"
            ]
        }
    },
    "additionalProperties": false,
    "required": [
        "status"
    ]
}
```

A.3.14. SendLocalList.

Este mensaje es de mucha utilidad a la hora de que el SAVE trabaje en modo off-line. Es decir que no tenga comunicación con el centro de carga pero de todos modos pueda continuar con su funcionamiento normal.

En estos casos en forma periódica (puede ser una vez al día) el centro de carga envía al SAVE (utilizando el mensaje SendLocalList) la lista de usuarios autorizados para realizar una carga y si en el momento que un usuario quiere cargar su VE el SAVE no tiene comunicación con el servidor puede chequear esa lista y permitir o no la carga.

Cabe destacar que en estos casos pude ser que el SAVE realice varios procesos de carga a diferentes usuarios antes de retomar la comunicación con el servidor. En estos casos cuando se retoma la comunicación el SAVE debe enviar todos los mensajes al servidor que debería de haber enviado en cada una de las cargas para que el gestor de carga reconstruya la comunicación correspondiente a cada sesión de carga.

A.3.14.1. SendLocalList Request.

La estructura del mensaje de tipo request es la siguiente.

```
{
"$schema": "http://json-schema.org/draft-04/schema#",
"title": "SendLocalListRequest",
"type": "object",
"properties": {
```

```
"listVersion": {
"type": "integer"
},
"localAuthorizationList": {
"type": "array",
"items": {
"type": "object",
"properties": {
    "idTag": {
        "type": "string",
        "maxLength": 20
    },
    "idTagInfo": {
        "type": "object",
"expiryDate": {
"type": "string",
"format": "date-time"
},
"parentIdTag": {
"type": "string",
"maxLength": 20
},
        "properties": {
            "status": {
                "type": "string",
                "enum": [
                    "Accepted",
                    "Blocked",
                    "Expired",
                    "Invalid",
                    "ConcurrentTx"
                ]
            }
        },
        "required": [
            "status"
        ]
    }
},
"required": [
    "idTag"
]
}
},
"updateType": {
"type": "string",
```

```
"enum": [
"Differential",
"Full"
]
}
},
"additionalProperties": false,
"required": [
"listVersion",
"updateType"
]
}
```

A.3.14.2. SendLocalList Response.

La estructura del mensaje de tipo response es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "SendLocalListResponse",
    "type": "object",
    "properties": {
        "status": {
            "type": "string",
            "enum": [
                "Accepted",
                "Failed",
                "NotSupported",
                "VersionMismatch"
            ]
        }
    },
    "additionalProperties": false,
    "required": [
        "status"
    ]
}
```

A.3.15. TriggerMessage.

Este mensaje solo se envía por parte del gestor de carga al SAVE. Se utiliza para decirle al SAVE cada que intervalos de tiempo el gestor de carga quiere recibir los siguientes mensajes; “BootNotification”, “DiagnosticsStatusNotification”, “FirmwareStatusNotification”, “Heartbeat”, “MeterValues” y “StatusNotification”.

Como ya se vio anteriormente, por ejemplo, el intervalo de tiempo que corresponde al mensaje “Heartbeat” el gestor de carga lo envía al momento de responder al mensaje “BootNotification” dentro del campo “interval”, pero si luego de esto el gestor de carga quiere cambiar ese intervalo de tiempo lo puede hacer a través del mensaje “TriggerMessage”. A continuación se muestra la estructura según el protocolo OCPP 1.6 para el mensaje “TriggerMessage” tanto para su forma request como response.

A.3.15.1. TriggerMessage Request.

La estructura del mensaje de tipo request según el protocolo OCPP 1.6 es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "TriggerMessageRequest",
    "type": "object",
    "properties": {
"requestedMessage": {
"type": "string",
"enum": [
"BootNotification",
"DiagnosticsStatusNotification",
"FirmwareStatusNotification",
"Heartbeat",
"MeterValues",
"StatusNotification"
]
},
        "connectorId": {
            "type": "integer"
        }
    },
    "additionalProperties": false,
    "required": [
        "requestedMessage"
    ]
}
```

A.3.15.2. TriggerMessage Response.

La estructura del mensaje de tipo response según el protocolo OCPP 1.6 es la siguiente.

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "title": "TriggerMessageResponse",
    "type": "object",
    "properties": {
        "status": {
            "type": "string",
            "enum": [
                "Accepted",
                "Rejected",
"NotImplemented"
            ]
        }
    },
    "additionalProperties": false,
    "required": [
        "status"
    ]
}
```

A.4. Instalación de NodeJS y npm.

Como se explicó en la sección 4.2.1 se optó por Node js por ser software libre y como tal es posible disponer de mucha infamación respecto a su uso y programación. En esta sección se presenta la forma de instalare Node js y npm.Para ellos basta ejecutar en la terminal de Linux el siguiente comando *apt-get install nodejs npm*. De esta forma se instalará nodejs y el gestor de paquetes npm. Una vez finalizada la instalación se tiene listo el entorno de desarrollo para comenzar a trabajar.

Una vez instalados NodeJS y el gestor de paquetes npm se debe crear una carpeta que contendrá todos los archivos y librerías del proyecto. Creada la carpeta es necesario instalar los paquetes que serán utilizados para programar el código del servidor js, para ello debe abrir la carpeta en una terminal de comandos y ejecutar *sudo npm install "Nombre del paquete"*, este comando deberá ser ejecutado cada vez que se pretenda instalar un paquete.

Una vez instalados los paquetes necesarios el código Java Script debe ser editado y guardado con extensión .js, implementado el código es posible verificar su funcionamiento ejecutando en la terminal de comandos *"node nombre del archivo.js"*.Para ello es necesario estar en la dirección de la carpeta creada para el proyecto. En el anexo B se podrá encontrar el código implementado en este caso.

A.5. Instalación del Controlador del CH340.

Dado que se está utilizando linux en su distribución Debian como sistema operativo en la Raspberry, para la instalación del controlador del CH340 fue necesario ejecutar el archivo *"ch34x.c"*, contenido en la carpeta *"Proyecto Save/CH341SER LINUX"* y dependiendo de la distribución de Linux instalado será necesario agregar en las dependencias que corresponda algunos archivos del sistema. Los mismos se pueden encontrar en el soporte digital que acompaña este documento en la carpeta *"Proyecto Save/Archivos del Sistema"*.

ANEXO B

CÓDIGO RASPBERRY.

B.1. Código Raspberry.

B.1.1. Save.js.

```
1
2  //*************************************************************************
3
4  // requiere the necesary archives
5  const WebSocket = require('ws');
6  const aleatorio = require ("./Id");
7  const Serialport = require('serialport');
8  const medidor = require ("./medidor");
9  const envio = require ("./mensaje");
10
11 var Authorize = require('./Authorize.json');
12 var BootNotification = require('./BootNotification.json');
13 var Heartbeat = require('./Heartbeat.json');
14 var MeterValues = require('./MeterValues.json');
15 var StartTransaction = require('./StartTransaction.json');
16 var StopTransaction = require('./StopTransaction.json');
17
18 // constant declaration
19 const url = 'ws://172.26.161.180/CentralSystemOCPP16J/WebSocketHandler.ashx/
      CP-GHM'; //'ws:localhost:8100'
20
21 // variables declaration
22 var port;
23 var socket;
24 var menssage_recived = {};
25 var mens;
26
27 var main_status = "E0";
28 var cont=0;
29 var arduino_data= undefined;
30 var tarjeta = undefined;
31
```

```
32 var main_Interval;
33 var Interval_response;
34 var timeout_reset;
35 var Interval_boot;
36 var Interval_latido;
37 var Interval_energy;
38
39 var time_main = 5000;
40 var time_close = 5000;
41 var time_meterValue = 60000;
42
43 var time_response_boot = 31000;
44 var time_response_authorize = 31000;
45 var time_response_startTransaction = 31000;
46 var time_response_StopTransaction = 31000;
47
48 var time_reset_authorize = 5*time_response_authorize;
49 var time_reset_startTransaction = 5*time_response_startTransaction;
50 var time_reset_StopTransaction = 5*time_response_StopTransaction;
51
52 ********** Main Program ***************
53 openSerialPort(); // calls the function that initializes communication with
      the Arduino
54
55 connect(); // calls the function that verifies connection and if it is OK,
      execute function "MAIN"
56
57 // Code that is executed by the event "data" (in the serial port), when
      Arduino sends a data
58 parser.on('data', function (data) {
59   arduino_data = data;
60   console.log('El Arduino envia a la Raspberry: ' + arduino_data);  // print
       the data_arduino in the terminal
61   console.log('                                    ');
62 });
63
64 // This is the MAIN function that handles communication with the server
65 function main(socket) {
66
67 socket.isAlive = true;
68 console.log('Me conecte bien al servidor' );
69 console.log('                              ' );
70
71 main_Interval = setInterval( function(){  // The main function is repeated
      completely, in each "time_main" Interval
72
73 console.log("                                                  ");
74 console.log("
      **********************************************************************")
      ;
75 console.log('  Corriendo programa CON servidor  ');
76 console.log("                                                  ");
77 console.log("Se esta ejecutando el estado: " + main_status);
78
79
80
```

```
81  switch(main_status){
82
83  case "E0":
84
85  // Bootnotification request
86
87  mens = "Notificacion";
88  envio.mensaje(mens,socket);
89  menssage_recived = {};
90  port.write('0');  // notify the Arduino that SAVE is NOT READY
91  main_status = "E1"; // go to next status, to read BootNotification response
92
93  break;
94
95  case "E1":
96
97  // Bootnotification response
98  if (menssage_recived[1] == BootNotification[1]){
99
100 console.log("Me llego un bootnotifiction response");
101 console.log(menssage_recived);
102
103 if (menssage_recived[2].status == "Accepted"){  //status bootnotification
       response
104
105 console.log("Entre a mandar el uno al Arduino");;
106 port.write('1');  // notify the Arduino to start looking for a vehicle
107 clearInterval(Interval_boot);
108 clearInterval(Interval_response);
109 Interval_response = undefined;
110 //console.log("Interval Response: " + Interval_response);
111
112 Interval_latido = setInterval(function(){
113 mens = "Latido";
114 envio.mensaje(mens,socket);
115 }, menssage_recived[2].interval*1000 );
116
117
118 if (StopTransaction[3].transactionId == 0 ){
119 // ANY transaction open
120 main_status = "E2"; // go to next status, to send Authorize request
121 arduino_data = undefined;
122 }else{
123 // There is a Transaction still OPEN !!
124 main_status = "E8";
125 }
126 }
127 else{  // Status BootNotification NOT accepted
128
129 if (Interval_boot == undefined){
130 Interval_boot = setInterval(function(){
131 main_status = "E0";  // go back to last status, to re-send Bootnotification
132 }, menssage_recived[2].interval*1000);
133 }
134 }
135
```

```
136 }
137 else{ // Not any new message from server
138
139 console.log("Interval_response: " + Interval_response );
140
141 if (Interval_response == undefined){
142 console.log("Entra al if de insteval response");
143 Interval_response = setInterval(function(){
144 main_status = "E0";  // go back to last status, to re-send Bootnotification
145 }, time_response_boot);  // time we hold server response
146 }
147 }
148
149 break;
150
151 case "E2":
152
153 if (arduino_data == 0){
154 arduino_data = undefined;
155 main_status = "E9";
156 }
157 else if ( (arduino_data != undefined) && (arduino_data != 1) ){ // Authorise
        request
158
159 tarjeta = arduino_data;
160 arduino_data = undefined;
161 mens = "Autorizacion";
162 envio.mensaje(mens,socket,tarjeta.slice(0,-1));
163 main_status = "E3";  // go to next status, to read Authorize response
164 }
165
166 break;
167
168 case "E3":
169
170 if (arduino_data == 0){
171 arduino_data = undefined;
172 main_status = "E2";
173 }
174 else if (menssage_recived[1] == Authorize[1]){ // Authorize response
175
176 console.log("Me llego un autorize response");
177 console.log(menssage_recived);
178
179 clearInterval(Interval_response);
180 clearTimeout(timeout_reset);
181 Interval_response = undefined;
182
183 if(menssage_recived[2].idTagInfo.status == "Accepted"){ // status Authorize
       response
184 console.log("La tarjeta es autorizada");
185 main_status = "E4";  // go to next status, to send start-transaction request
186 }
187 else{
188 console.log("No se autorizo la tarjeta ");
189 port.write('4');  //52 for arduino
```

```
190 arduino_data = undefined; // clear arduino data
191 main_status = "E2";  // go back to last status, to read another card and re-
    send Authorize request
192 }
193
194 }
195 else{
196 if (Interval_response == undefined){
197 Interval_response = setInterval(function(){
198 main_status = "E2";  // go back to last status, to re-send Bootnotification
199 }, time_response_authorize);  // time we hold server response
200
201 timeout_reset = setTimeout(function() {
202 clearInterval(Interval_response);
203 main_status = "E0";
204 }, time_reset_authorize );
205 }
206 }
207
208 break;
209
210 case "E4":
211
212 if (arduino_data == 0){
213 arduino_data = undefined;
214 main_status = "E2";
215 }
216 else {       // Send StartTransaction request
217 medidor.energia(function(resultado){
218 // console.log("Dato del medidor: " + resultado);
219
220 StartTransaction[3].idTag = tarjeta.slice(0,-1);  // set value on JSON
    StartTransaction
221 mens = "ComenzarTransaccion";
222 envio.mensaje(mens,socket,resultado[0]*1000);
223 });
224
225 main_status = "E5";  // go to next status, to read Authorize response
226 }
227 break;
228
229 case "E5":
230
231 if (menssage_recived[1] == StartTransaction[1]){  // Read StartTransaction
    response
232
233 console.log("Llego un StartTransaction response");
234 console.log(menssage_recived);
235
236 clearInterval(Interval_response);
237 clearTimeout(timeout_reset);
238 Interval_response = undefined;
239
240 if (menssage_recived[2].idTagInfo.status == "Accepted") { // status
    StartTransaction response
241 console.log("StartTransaction Accepted");
```

```
242 console.log("Envio un dos al Arduino");
243 port.write('2');    // Tell Arduino to start Transaction
244
245 StopTransaction[3].transactionId = menssage_recived[2].transactionId;
246
247 // send MeterValue each time_metervalue
248 Interval_energy = setInterval(function(){
249 medidor.energia(function(resultado){
250 // console.log("Dato del medidor: " + resultado);
251 mens = "MandarValor";
252 envio.mensaje(mens,socket,resultado[0]*1000);
253 });
254 }, time_meterValue ); // send MeterValue each time_metervalue
255
256 main_status = "E6";
257 }
258 else{
259 console.log("StartTransaction Rejected");
260 port.write('5');  // StartTransaction NOT accepted, tell ARDUINO and put it
       on DISPLAY
261 arduino_data = undefined; // clear arduino data
262 main_status = "E2";  // go back to read another card and re-send Authorize
       request
263 }
264 }
265 else {
266 if (Interval_response == undefined){
267 Interval_response = setInterval(function(){
268 main_status = "E4";  // go back to last status, to re-send StartTransaction
269 }, time_response_startTransaction);  // time we hold server response
270
271 timeout_reset = setTimeout(function() {
272 clearInterval(Interval_response);
273 main_status = "E0";
274 }, time_reset_startTransaction );
275 }
276 }
277
278 break;
279
280 case "E6":
281
282 console.log(" Dato Tarjeta: " + tarjeta);
283 console.log(" Dato arduino: " + arduino_data);
284
285 // send StopTransaction Request
286 if (arduino_data == 0){
287
288 main_status = "E8";
289 }
290 else if( (arduino_data == tarjeta) || (arduino_data == 3)  ){ // Wait "
       tarjeta" and send StopTransaction request
291 // OR arduino tells "Stop": Vehicle IS DISCONNECTED (12V)
292 clearInterval(Interval_energy);
293 medidor.energia(function(resultado){
294 mens = "DetenerTransaccion";
```

```
295 envio.mensaje(mens,socket,resultado[0]*1000);
296 });
297
298 arduino_data = undefined;
299 main_status = "E7"; // go to wait for StartTransaction Response
300 }
301 else if(arduino_data != undefined) {
302 arduino_data = undefined;
303 port.write('6'); // Tell Arduino that (Arduino_data != tarjeta)
304 }
305 break;
306
307 case "E7":
308
309 if (menssage_recived[1] == StopTransaction[1]){ // Read a StopTransaction
       Response
310
311 console.log("Me llego un StopTransaction response");
312 console.log(menssage_recived);
313
314 clearInterval(Interval_response);
315 clearTimeout(timeout_reset);
316 Interval_response = undefined;
317
318 if (menssage_recived[2].idTagInfo.status == "Accepted") {
319 console.log("StopTransaction Accepted");
320 port.write('3'); // Tell Arduino to STOP charge
321 StopTransaction[3].transactionId = 0;  // Reset value "meterStop" to know
       the Transaction is CLOSED
322 arduino_data = undefined;
323 main_status = "E2"; // restart and wait a new "tarjeta" for a new charge
324 }
325 else{
326 console.log("UTE no permite parar la carga");
327 arduino_data = undefined;
328 main_status = "E6";
329 }
330 }
331 else{
332 if (Interval_response == undefined){
333 Interval_response = setInterval(function(){
334 main_status = "E8";  // go back to last status, to re-send StopTransaction
335 }, time_response_StopTransaction);  // time we hold server response
336
337 timeout_reset = setTimeout(function() {
338   port.write('0');  // tell Arduino there is an Error
339 }, time_reset_StopTransaction );
340 }
341 }
342
343 break;
344
345 case "E8":
346 // send StopTransaction Request while there is a ERROR in Arduino
347 clearInterval(Interval_energy);
348
```

```
349 medidor.energia(function(resultado){
350 mens = "DetenerTransaccion";
351 envio.mensaje(mens,socket,resultado[0]*1000);
352 });
353
354 main_status = "E7";
355 // tarjeta = undefined;
356 arduino_data = undefined;
357
358 break;
359
360 case "E9":
361
362 console.log("Estado error definitivo");
363 clearInterval(Interval_latido);
364 clearInterval(Interval_energy);
365
366 if (arduino_data == 1){ main_status = "E0"};
367
368 break;
369
370
371 default:
372
373 console.log("El programa principal no entra en ningun estado de (main_state)
    ");
374
375 } // Finish the switch(main_status)
376
377
378
379
380 // print in the log, if the websocket is still active or not
381 console.log('Socket is Alive? ' + socket.isAlive);
382 console.log('            ');
383 if (socket.isAlive==false){
384 return socket.close(1000); } //terminate();}       //  }
385
386 socket.isAlive =false;
387 socket.ping(function(){});
388
389 }, time_main); //  The main function is repeated completely, in each "
    time_main" Interval
390
391 }
392
393
394 // ********************************
395 //  ******  Functions used  *******
396 // ********************************
397
398 //  Function that verifies the state of the connection, and if it is OK, it
    executes function "main"
399 function connect() {
400 console.log("entre al conect");
401 socket = new WebSocket(url,'ocpp1.6');    // Open a NEW SOCKET
```

```
402
403 socket.on('pong', function(){this.isAlive=true;});  // check if the socket
       between UTE and SAVE is active or not
404
405 // Execute this line, if the connection was opened OK
406 socket.on('open', function(){
407 main(socket);                //  RUN THE "MAIN" FUNCTION !!!
408 console.log("entre al open");
409 cont = 0;     //  Reset the counter, used by the "connect" function
410 });
411
412 // It works when a message arrives
413 socket.on('message', function incoming(stringJson){
414 menssage_recived = JSON.parse(stringJson || '{}' );
415 console.log('NEW message from server');
416 });
417
418 // Works when there is NO connection
419 socket.on('error', function error(error){
420 console.log(Error de conexion);
421 console.log('                ');
422 });
423
424 //  Execute these lines if the connection is closed or if the websocket is
       not opened
425 socket.on('close',function(){
426 clearInterval(main_Interval);
427 console.log("entre al close");
428 setTimeout(function() {
429 socketClose()        // calls the function "socketClose", defined at the end
        of this file
430 }, time_close );
431 });
432
433 }  // the declaration of the "connect" function ends
434
435
436 // Function that initializes communication with the Arduino
437 function openSerialPort(){
438
439 const Readline = Serialport.parsers.Readline;//le pongo esa propiedad"parser
       " que es para que las lecturas se vean mejor y las muestre en una sola
       linea.
440 port = new Serialport('/dev/ttyACM0', {baudRate: 115200 //para q trasmitan a
        la misma velocidad.
441 }) ;
442 parser = port.pipe(new Readline({ delimeter: '\r\n'}));
443
444 }
445
446
447 // Function called by the main function "Connect" ...
448 //  And controls in state with server disconnected
449 function socketClose(){
450 cont = cont + 1;
451
```

```
452 if (cont == 10){                //  if I pass a time (cont = 10); cut the VE
       load and then close the socket
453 console.log('Se ejecuta (cont == 10) del SocketClose... ');
454 console.log('                                          ');
455 clearInterval(Interval_energy);
456 clearInterval(Interval_latido);
457 clearInterval(Interval_response);
458 Interval_response = undefined;
459 port.write('0');  // send code Error to Arduino
460 main_status = "E0";
461 }
462 console.log('El valor del contador de socketClose es: ' + cont);
463 connect();
464 }
```

Listing B.1: Save.js

B.1.2. Medidor.js.

```
1
2  var ModbusRTU = require("modbus-serial");
3  var client = new ModbusRTU();
4  var hexToBinary = require("hex-to-binary");
5
6  console.log("Se inicio comunicacion con el medidor");
7
8
9  client.connectRTU("/dev/ttyUSB0", { baudRate:9600}); // /dev/ttyUSB0
10
11
12 function energia(callback) {
13
14 console.log("Entre a mandar la energia");
15 client.setID(1);
16
17 // read the 2 registers starting at address 35
18 // on device number 1.
19
20 client.readHoldingRegisters(71,8).then( (energy) => {
21
22 const buf = energy.buffer;
23 var E= [];
24 var i= 0 ;
25 var j= 0 ;
26 while (i<=12){
27 E[j] = buf.readFloatBE(i)/1000;
28 i= i+4;
29 j= j+1;
30 }
31
32 callback(E);
33 });
34
35 }
36
37 // var U= [] ;
```

```
// var I= [] ;
// var P= [] ;
// var Q= [] ;
var PF= [] ;
// var S= [] ;
var F;

function frecuencia_y_FP(callback) {
client.readHoldingRegisters(54,9).then((salida2) => {

for (var i= 0 ; i<9 ;i++){
if (i<4) {
PF[i]= (salida2.data[i]/10**3);
}
else if (i == 8){
F = salida2.data[8]/10**2;
}
}

var salida = [F,PF[0]];
callback(salida);
//console.log('Factor de Potencia');
//console.log(PF);
//console.log('Frecuencia');
//console.log(F);
//console.log('Potencia Aparente');
//console.log(S);
});

console.log("Mando la frecuencia el el factor de potencia");
}

module.exports = {
"energia":energia,
"frecuencia_y_FP":frecuencia_y_FP
}
```

Listing B.2: Medidor.js

B.1.3. mensaje.js

```

// importo los formatos pre-determinados de cada mensaje JSON de request al
   servidor

var Authorize = require('./Authorize.json');
var BootNotification = require('./BootNotification.json');
var Heartbeat = require('./Heartbeat.json');
var MeterValues = require('./MeterValues.json');
var StartTransaction = require('./StartTransaction.json');
var StopTransaction = require('./StopTransaction.json');

var DataTransfer = require('./DataTransfer.json');
var DiagnosticsStatusNotification = require('./DiagnosticsStatusNotification
   .json');
```

```
13 var FirmwareStatusNotification = require('./FirmwareStatusNotification.json
   ');
14 var GetLocalListVersion = require('./GetLocalListVersion.json');
15 var SendLocalList = require('./SendLocalList.json');
16 var StatusNotification = require('./StatusNotification.json');
17 var UpdateFirmware = require('./UpdateFirmware.json');
18 var Energia = require('./Energia.json');
19
20 const aleatorio = require ("./Id"); // se utiliza funcion para generar un Id
    aleatorio para cada mensaje JSON
21
22 //var fecha= new Date(); var fecha = Date.now();
23 //console.log("Entre a enviar mensaje...Muestro el mensaje que yo envio");
24
25 function mensaje(tipo,socket,dato){
26
27 var Fecha_Hora = new Date();
28 console.log(" Muestro el mensaje que estoy por enviar... ");
29 console.log('                          ');
30
31 switch (tipo)  {
32
33 case "Autorizacion":
34 Authorize[1] = aleatorio.Id(); //este ID aleatorio hay que ponerlo en todos
   los mensajes que voy a enviar.
35 Authorize[3].idTag = dato;
36 socket.send( JSON.stringify(Authorize) );  // Envio al servidor el objeto
   Json como un string
37 console.log(Authorize);
38 console.log('                          ');
39 break;
40
41 case "Notificacion":
42 BootNotification[1] = aleatorio.Id();
43 socket.send( JSON.stringify(BootNotification) );  // Envio al servidor el
   objeto Json como un string
44 console.log(BootNotification);
45 console.log('                          ');
46 break;
47
48 case "Latido":
49 Heartbeat[1] = aleatorio.Id();
50 socket.send( JSON.stringify(Heartbeat) );  // Envio al servidor el objeto
   Json como un string
51 console.log(Heartbeat);
52 console.log('                          ');
53 break;
54
55 case "MandarValor":
56 MeterValues[1] = aleatorio.Id();
57 MeterValues[3].meterValue[0].sampledValue[0].value = dato;
58 MeterValues[3].meterValue[0].timestamp = Fecha_Hora;
59 socket.send( JSON.stringify(MeterValues) );  // Envio al servidor el objeto
   Json como un string
60 console.log( JSON.stringify(MeterValues) );
61 console.log('                          ');
```

```
62  break;
63
64  case "Energia":
65  Energia[1] = aleatorio.Id();
66  Energia[3].meterValue[0].sampledValue[0].value = dato;
67  socket.send( JSON.stringify(Energia) );  // Envio al servidor el objeto Json
        como un string
68  console.log( JSON.stringify(Energia) );
69  console.log('                          ');
70  break;
71
72  case "ComenzarTransaccion":
73  StartTransaction[1] = aleatorio.Id();
74  StartTransaction[3].meterStart = dato;
75  StartTransaction[3].timestamp = Fecha_Hora;//No se si esto ya esta entre
       comillas o no...Compare con el ejemplo que el loco nos mando y vi que no
       aparecen las comillas en los tags que envimos nosotros,,...ver eso.
76  socket.send( JSON.stringify(StartTransaction) );  // Envio al servidor el
       objeto Json como un string
77  console.log(StartTransaction);
78  console.log('                          ');
79  break;
80
81  case "DetenerTransaccion":
82  StopTransaction[1] = aleatorio.Id();
83  StopTransaction[3].meterStop = dato;
84  StopTransaction[3].timestamp = Fecha_Hora;
85  socket.send( JSON.stringify(StopTransaction) );  // Envio al servidor el
       objeto Json como un string
86  console.log(StopTransaction);
87  console.log('                          ');
88  break;
89
90  ////////////////////////////
91  Mensajes que no tenemos implementados en el programa principal:
92  ////////////////
93
94  case "EnviarListaLocal":
95  SendLocalList[1] = aleatorio.Id();
96  socket.send( JSON.stringify(SendLocalList) );  // Envio al servidor el
       objeto Json como un string
97  break;
98
99  case "EstadoNotificacion":
100 StatusNotification[1] = aleatorio.Id();
101 StatusNotification[3].timestamp = Fecha_Hora;
102 StatusNotification[3].info = "Available for new transaction",//estas cosas
       despues van a cambiar dependiendo de la situacion en la que se este.
103 StatusNotification[3].status = "Available",
104 StatusNotification[3].errorCode = "NoError";
105 socket.send( JSON.stringify(StatusNotification) );  // Envio al servidor el
       objeto Json como un string
106 break;
107
108 case "ObtenerListaLocal":
109 GetLocalListVersion[1] = aleatorio.Id();
```

```
110 socket.send( JSON.stringify(GetLocalListVersion) );  // Envio al servidor el
        objeto Json como un string
111 break;
112
113 case "DiagnosticoEstadoNotificacion":
114 DiagnosticsStatusNotification[1] = aleatorio.Id();
115 socket.send( JSON.stringify(DiagnosticsStatusNotification) );  // Envio al
       servidor el objeto Json como un string
116 break;
117
118 case "TransferirDatos":
119 DataTransfer[1] = aleatorio.Id();
120 socket.send( JSON.stringify(DataTransfer) );  // Envio al servidor el objeto
        Json como un string
121 break;
122
123 default:
124 console.log('No se envio el mensaje al servidor, ya que ese pedido de
       mensaje no existe...');
125 console.log('                                                    ');
126 }
127
128 }
129
130 module.exports = {
131 "mensaje":mensaje
132 }
```

Listing B.3: mensaje.js.

B.1.4. Id.js.

```
1
2 function Id(){           //codigo para crear el ID del mensaje de forma
      aleatoria.
3    var text = "";
4    var possible =
         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
5    for (var i = 0; i < 20; i++)
6    text += possible.charAt(Math.floor(Math.random() * possible.length));
7    return text;
8    }
9
10 module.exports = {
11
12 "Id":Id
13 }
```

Listing B.4: Id.js

B.1.5. Authorize.json.

```
[
    2,
    "123321",
    "Authorize",
    {
      "idTag":"8BC57123"
    }
]
```

B.1.6. Energia.json.

```
[
  2,
  "B89AD21C-D300-11E8-D300-11E8B9CC078B",
  "MeterValues",
  {
    "connectorId":1,
    "meterValue":
      [
        {
        "sampledValue":
          [
            {
            "measurand":"Energy.Active.Import.Register",
            "unit":"Wh",
            "value":"250.000000"
            },
            {
            "measurand":"Power.Active.Import",
            "unit":"W",
            "value":"1460.000000"
            }
          ],
        "timestamp":"2018-10-18T18:07:45Z"
        }
      ]
  }
]
```

B.1.7. BootNotification.json.

```
[
    2,
    "87744D02-9CF0-11E9-9CF0-11E989BFE9EE",
    "BootNotification",
    {
      "chargePointModel":"FinDeCarrera",
      "chargePointVendor":"GutierrezHaltyMango"
    }
]
```

B.1.8. DataTransfer.json.

```
[
    2,
    "123456",
    "DataTransfer",
    {
      "vendorId":"string"
    }
]
```

B.1.9. Heartbeat.json.

```
[
  2,
  "1233322PPWDFS",
  "Heartbeat",
  {

  }
]
```

B.1.10. MeterValues.json.

```
[
  2,
  "B89AD21C-D300-11E8-D300-11E8B9CC078B",
  "MeterValues",
  {
    "connectorId":1,
    "meterValue":
    [
      {
        "sampledValue":
        [
          {
      "measurand":"Energy.Active.Import.Register",
      "unit":"Wh",
          "value":"250.000000"
          }
        ],
    "timestamp":"2018-10-18T18:07:45Z"
      }
    ]
  }
]
```

B.1.11. StartTransaction.json.

```
[
  2,
  "numoerocontrol",
  "StartTransaction",
  {
    "connectorId":1,
    "idTag": "",
    "meterStart":2323,
    "timestamp":"2020-01-24T17:41:00.5973677-03:00"
  }
]
```

B.1.12. StopTransaction.

```
[
  2,
  "sdfsdfs23232323",
  "StopTransaction",
  {
    "transactionId":0,
    "timestamp":"2013-02-01T20:53:32.486Z",
    "meterStop":0
  }
]
```

B.1.13. DiagnosticsStatusNotification.json.

```
[
  2,
  "23324f",
  "DiagnosticsStatusNotification",
  {
    "status":"uploaded"
  }
]
```

B.1.14. FirmwareStatusNotification.json.

```
[
  2,
  "2423",
  "FirmwareStatusNotification",
  {
    "status":"downoader"
  }
]
```

B.1.15. GetLocalListVersion.json.

```
[
  2,
  "234324",
  "GetLocalListVersion",
  {

  }
]
```

B.1.16. SendLocalList.json.

```
[
  2,
  "34234",
  "SendLocalList",
  {
    "listVersion":2344,
    "updateType":"differential"
  }
]
```

B.1.17. UpdateFirmware.json.

```
[
  2,
  "sdf4234324",
  "UpdateFirmware",
  {
    "location":"sdfsdf",
    "retrieveDate":"2013-02-01T20:53:32.486Z"
  }
]
```

ANEXO C

CÓDIGO ARDUINO.

C.1. Código Arduino.

C.1.1. Save.ino.

```
#include "Save.h"

void setup() {

  Save.Init();
}

void loop(){

    Save.Read();

    Save.StateMain();

    Save.StateDisplay();

    // delay(300);
}
```

C.1.2. Save.h.

```
#ifndef Save_h
#define Save_h

#include <Arduino.h>

#define State_A0      0
#define State_A1      1
#define State_A2      2
#define State_A3      3
#define State_A4      4
#define State_A5      5
#define State_A6      6

////////////////////////////////

#define VoutMin_A1        11.5  // Votlts
#define VoutMax_A1        12.5

#define VoutMin_B1        8.5  // Votlts
#define VoutMax_B1        9.5

#define VoutMin_C1        5.5  // Votlts
#define VoutMax_C1        6.5

////////////////////////////////

#define DispState_A0     0
#define DispState_A1     1
#define DispState_A2     2
#define DispState_B1     3
#define DispState_C1     4
#define DispState_C2     5
#define DispState_D1     6
#define DispState_D2     7
#define DispState_E1     8
#define DispState_F1     9
#define DispState_F2     10
#define DispState_F3     11
#define DispState_G1     12
#define DispState_H1     13
#define DispState_I1     14
#define DispState_I2     15
#define DispState_J1     16

////////////////////////////////

#define Pin_pwm          3
#define Start            5
#define Error            0
#define Stop             3
#define Restart          1

#define RST_PIN          9     //Pin 9 para el reset del RC522
#define SS_PIN           10     //Pin 10 para el SS (SDA) del RC522
```

```
55
56 #define CodigoError     1
57
58 ////////////////////////////////
59
60 class SaveClass
61 {
62   public:
63         SaveClass();  // Constructor
64
65         void Init();  // Method
66
67         void Read();  // Method
68
69         void StateMain(); //  Method
70
71         void StateDisplay(); //  Method
72
73   private:
74                         // Privated and use by StateMain();
75         void StateA0();
76
77         void StateA1();
78
79         void StateA2();
80
81         void StateA3();
82
83         void StateA4();
84
85         void StateA5();
86
87         void StateA6();
88
89         bool ReadCard();
90
91         int DutyCicle();
92
93                       // Privated and use by StateDysplay();
94
95         void message(String FirstMessage, String SecondMessage);
96
97
98                                     // Privated variables
99         uint8_t DispStateValue = DispState_A0;
100
101        uint8_t StateValue = State_A0;
102
103 };
104
105 extern SaveClass Save;
106
107 #endif
```

C.1.3. Save.cpp.

```
1
2 ////////////////////////////////////
3
4 #include "Save.h"
5
6 #include <Arduino.h>
7 #include <PWM.h>
8 #include <SPI.h>
9 #include <MFRC522.h>
10 #include <LiquidCrystal_I2C.h>
11
12 ////////////////////////////////////
13
14 LiquidCrystal_I2C lcd(0x3f,16,2);
15
16 MFRC522 mfrc522(SS_PIN, RST_PIN); //Creamos el objeto para el RC522
17
18 ////////////////////////////////////////////
19 float Vout = 0;
20 int AnalogVout = 0;
21 int32_t frequency = 1000;
22
23 int cicle = 0;
24 int analogWire_value = 0;
25 int StatusCard = 0;
26 int ErrorConnection = 0;
27
28 long TimeError = 0;
29 long TimeCard =0;
30 int DelayCard = 30000;
31 int DelayError = 30000;
32 long TimeDisp = 0;
33 int Delay = 5000;
34
35 long TimeRead = 0;
36 int DelayRead = 20;
37
38 ////////////////////////////////////
39
40 float Wire_value  = 0;
41 float Wire_Max = 4.77;
42 float Wire_Max_13 = 4.67;
43 float Wire_Min_13 = 4.24;
44 float Wire_Max_20 = 4.05;
45 float Wire_Min_20 = 3.23;
46 float Wire_Max_32 = 2.92;
47 float Wire_Min_32 = 2.14;
48 float Wire_Min = 1.07;
49 float Wire_Max_63 = 1.94;
50 float Wire_Min_63 = 1.33;
51
52 ////////////////////////////////////////////////
53
54 SaveClass::SaveClass(){
```

```
55
56   }
57
58 ////////////////////////////////////////////////
59 //esta fuci n es llamda en setup para inivializar parametros y variables
60 void SaveClass::Init(){
61
62     // put your setup code here, to run once:
63   Serial.begin(115200); // Iniciamos la comunicaci n  serial
64   SPI.begin();          // Iniciamos el Bus SPI
65   mfrc522.PCD_Init(); // Iniciamos  el MFRC522
66
67   // Serial.println("Lectura del UID");
68   lcd.init();
69   lcd.backlight();
70   lcd.clear();
71
72   pinMode(Pin_pwm,OUTPUT);  //pines de 1 kHz
73   pinMode(Start,OUTPUT);
74   pinMode(A3,INPUT);
75   pinMode(A2,INPUT);
76   digitalWrite(Start, HIGH);
77   InitTimersSafe();
78   SetPinFrequencySafe(Pin_pwm,frequency);
79
80 }
81
82 ////////////////////////////////////////////////
83
84 //function of states
85 void SaveClass::StateMain() {
86
87   switch (StateValue) {
88
89     case State_A0:  StateA0();
90     break;
91     case State_A1:  StateA1();
92     break;
93     case State_A2:  StateA2();
94     break;
95     case State_A3:  StateA3();
96     break;
97     case State_A4:  StateA4();
98     break;
99     case State_A5:  StateA5();
100    break;
101    case State_A6:  StateA6();
102    break;
103  }
104 }
105
106 ////////////////////////////////////////////////////////////////
107
108 void SaveClass::StateA0(){
109
110   //Serial.println("StateInit");
```

```
111   //2Serial.println((char)Serial.read());
112   if (Serial.read() == 49){      // ascci 49
113
114     if ( Vout > VoutMin_A1 && Vout < VoutMax_A1){
115       StateValue=State_A1;       // state 12V
116       }
117       else {
118               DispStateValue = DispState_I1; // Error screen; Agregar estado de
                    Error
119               StateValue = State_A6;
120               Serial.println(Error);
121             }
122   }
123    else  DispStateValue = DispState_A0;     // starting screen (Fuera de servicio
        )
124 }
125
126 ////////////////////////////////////////////////
127
128 void SaveClass::StateA1(){
129
130   ErrorConnection = Serial.read();
131
132     if (ErrorConnection != 48){
133
134         if ((Vout > VoutMin_A1 && Vout < VoutMax_A1) || (Vout > VoutMin_B1 &&
              Vout < VoutMax_B1)) {
135
136           bool CardValue = ReadCard();
137           // Serial.println(CardValue);
138
139           if (CardValue == true) {
140                 StateValue = State_A2;         //state 12V o 9V
141                 TimeCard = millis();
142             }
143             else if ((DispStateValue != DispState_A1) && (DispStateValue !=
                    DispState_A2)) {
144                           DispStateValue = DispState_A2;  // screen Pass card
145                           //Serial.println("3");
146                       }
147         }
148           else {
149                   DispStateValue = DispState_I1; // Error screen; (connector
                      error)
150                   StateValue = State_A6;
151                   Serial.println(Error);
152                 }
153     }
154           else {
155                  digitalWrite(Pin_pwm, LOW);
156                  delay(100);
157                  digitalWrite(Start, HIGH);
158                  StateValue = State_A0;        // starting screen state (Out of
                     service)
159                 }
160 }
```

```
161
162 ////////////////////////////////////////////
163
164 // State 6 V which vehicle ready for loading and waiting  load start
165 void SaveClass::StateA2(){
166
167       //  Serial.println("StateB1");
168   //if (Vout > VoutMin_A1 && Vout < VoutMax_A1)) {    //stop charging if while
        waiting for a card the vehicle is disconnected.
169
170   if (millis() > TimeCard + DelayCard) {
171
172       Serial.println(Error);
173       StateValue = State_A6;
174       DispStateValue = DispState_A0;
175       // TimeCard = millis();
176       digitalWrite(Pin_pwm, LOW);
177       delay(100);
178       digitalWrite(Start,HIGH);
179     }
180
181      else {
182             StatusCard = Serial.read();
183
184             if (StatusCard != 48) {
185
186               if (StatusCard == 52) {   // waits a 4
187                         DispStateValue = DispState_E1;    //Invalid card
188                         StateValue = State_A1;            //state 12V o 9V
                                  hay que ver que hacer en el caso que la
                              tarjeta sea invalida para detener la carga
189                 }
190                 else if ( StatusCard == 51) {    // waits a 3
191                         //  Serial.println("Stop_load");
192                           digitalWrite(Pin_pwm, LOW);
193                           delay(100);
194                           digitalWrite(Start,HIGH);          //stop load
195                           StateValue = State_A5;
196                         }
197                         else if (StatusCard == 50) { // waits a 2
198                               StateValue = State_A3;
199                             }
200                             else if (StatusCard == 53) {  // waits a 5
201                                       DispStateValue = DispState_J1;
202                                       StateValue=State_A5;
203                                   }
204                                   else if (StatusCard == 54) {   // waits a 6
205                                          DispStateValue = DispState_E1;    //
                                              Invalid card
206                                          StateValue = State_A4;            //
                                              state 12V o 9V
207                                    }
208                                    else  if ((DispStateValue !=
                                         DispState_D1) && (DispStateValue !=
                                          DispState_D2)) {
209                                           DispStateValue = DispState_D1;
```

```
210                                                 }
211                                             /* }
212                                               else {StateValue = State_A0;
213                                               } */
214             }
215               else { StateValue = State_A4; }
216         }
217 }
218
219 ////////////////////////////////////////////
220
221 void SaveClass::StateA3(){
222
223   ErrorConnection = Serial.read();
224
225   if (ErrorConnection != 48) {
226
227       if (Vout > VoutMin_B1 && Vout < VoutMax_B1) {  // 9V
228
229           cicle = DutyCicle();
230
231           if (cicle != CodigoError) {
232             digitalWrite(Start,LOW);  // Start load
233             delay(100);
234             pwmWrite(Pin_pwm,cicle); //Start pwm
235             StateValue = State_A4;
236           }
237             else {
238                     DispStateValue = DispState_I1;  // Erros Screen (connection
                           error)
239                     StateValue = State_A6;
240                     Serial.println(Error);
241                     delay(50);
242                   }
243
244       }
245         else { DispStateValue = DispState_B1; }
246
247   }
248     else { StateValue = State_A0; }
249
250 }
251
252 ////////////////////////////////////////////
253
254 // State loading
255 void SaveClass::StateA4() {
256
257   ErrorConnection = Serial.read();
258
259   if (ErrorConnection != 48) {
260
261         if  (Vout < VoutMin_A1 || Vout > VoutMax_A1) {    // Only when Vout != 12
                V
262
263           bool CardValue = ReadCard();
```

```
264             // Serial.println(CardValue);
265
266             if (CardValue == true) {
267                   StateValue = State_A2;
268                   TimeCard = millis();
269             }
270             else if ((DispStateValue != DispState_F1) && (DispStateValue !=
                    DispState_F2) && (DispStateValue != DispState_F3)) {
271                     DispStateValue = DispState_F1; // screen Pass card
272                     //Serial.println("3");
273                   }
274
275         }
276
277         else {                                  // Only when Vout == 12V
278                 digitalWrite(Pin_pwm, LOW);
279                 delay(100);
280                 digitalWrite(Start,HIGH);   // stop load
281                 Serial.println(Stop);
282                 StateValue = State_A1;
283               }
284   }
285     else {                                // Only ErrorConnection == 48
286           digitalWrite(Pin_pwm, LOW);
287           delay(100);
288           digitalWrite(Start,HIGH);
289           StateValue = State_A0;
290          }
291 }
292
293 ////////////////////////////////////////////
294
295 void SaveClass::StateA5(){
296
297   ErrorConnection = Serial.read();
298
299   if (ErrorConnection != 48) {
300
301           // Serial.println("State_A3");
302                 if  (Vout > VoutMin_A1 && Vout < VoutMax_A1) {
303                       StateValue = State_A1;
304                 }
305                  else if( DispStateValue != DispState_J1 ) {
306                             DispStateValue = DispState_G1;
307                           }
308   }
309     else {
310           digitalWrite(Pin_pwm, LOW);
311           delay(100);
312           digitalWrite(Start,HIGH);
313           StateValue = State_A0;
314         }
315
316 }
317
318 ////////////////////////////////////////////
```

```
319 // function state error
320 void SaveClass::StateA6(){
321
322   ErrorConnection = Serial.read();
323
324   if (ErrorConnection != 48) {
325
326       if (Vout > VoutMin_A1 && Vout < VoutMax_A1) {
327                 StateValue = State_A1;
328                 Serial.println(Restart);
329             }
330               else if (millis() > TimeError + DelayError) {
331                       Serial.println(Error);
332                       TimeError  = millis();
333                 }
334                 /*  else   if ( Vout> VoutMin_B1 && Vout < VoutMax_B1 ) {
335                         StateValue = State_A2;
336                       }
337                       else   if ( Vout> VoutMin_C1 && Vout < VoutMax_C1 ) {
338                             StateValue = State_A4;
339                          }*/
340   }
341       else {
342                 digitalWrite(Pin_pwm, LOW);
343                 delay(100);
344                 digitalWrite(Start,HIGH);
345                 StateValue = State_A0;
346             }
347
348 }
349
350
351 ////////////////////////////////////////
352    // this function reads the card
353 bool SaveClass::ReadCard(){
354
355   if ( mfrc522.PICC_IsNewCardPresent() ) {
356
357       //Seleccionamos una tarjeta
358       if ( mfrc522.PICC_ReadCardSerial()) {
359
360           // Enviamos serialemente su UID
361           // Serial.print("Card UID:");
362               for (int i = 0; i < mfrc522.uid.size; i++) {
363                   Serial.print(mfrc522.uid.uidByte[i],HEX);
364               }
365
366               Serial.println();
367           // Terminamos la lectura de la tarjeta  actual
368               mfrc522.PICC_HaltA();
369               return true;
370        }
371
372   }
373
374   return false;
```

```
375 }
376
377 ////////////////////////////////////
378     // this function calculates the duty cycle
379 int SaveClass::DutyCicle(){
380
381     analogWire_value = analogRead(A2); // pin donde se lee el cable
382
383     Wire_value = ( analogWire_value * ( 5.0/ 1023.0 ) );
384
385     if ((Wire_value >  Wire_Max)) {
386       return CodigoError;
387     }
388
389     if ((Wire_value < Wire_Max_13) && (Wire_value > Wire_Min_13 )) {
390       //la corriente maxima es de 13A segun el cable
391       return  255 - 2.55 * (13 / 0.6);
392     }
393
394     if ((Wire_value < Wire_Max_20) && (Wire_value > Wire_Min_20)) {
395       //la corriente maxima es de 20A segun el cable
396       return  255 - 2.55*(20 / 0.6);
397     }
398
399     if ((Wire_value < Wire_Max_32) && (Wire_value> Wire_Min_32)) {
400       //la corriente maxima es de 32A segun el cable
401       return 255 - 2.55 * (32 / 0.6);
402     }
403
404     if ((Wire_value < Wire_Max_63) && (Wire_value > Wire_Min_63)) {
405       //la corriente maxima es de 63A segun el cable... pero limitan mis 32A
406       return   255 - 2.55* (32 / 0.6);
407     }
408
409     if ((Wire_value < Wire_Min)) {
410       return CodigoError;
411     }
412
413 }
414
415 ////////////////////////////////
416
417 ////////////////////////////////
418
419 void SaveClass::Read() {
420
421   //Serial.println("Ejecuta el Read");
422
423   TimeRead = millis();
424   // Serial.println(Vout);
425   // Serial.println(TimeRead);
426
427   while ( millis() - TimeRead < DelayRead ) {
428
429       AnalogVout = analogRead(A3);
430       Vout = (AnalogVout * (5.0/1023.0)*3 + 0.7 );
```

```
431       //    Serial.println(Vout);
432
433       if ( Vout > VoutMin_A1 && Vout < VoutMax_A1) {
434            //       Serial.println(Vout);
435         return;}
436                 else if ( Vout > VoutMin_B1 && Vout < VoutMax_B1) {
437                   //       Serial.println(Vout);
438                       return; }
439                               else if ( Vout > VoutMin_C1 && Vout < VoutMax_C1)
                                    {
440                                 //  Serial.println(Vout);
441                                     return; }
442   }
443
444 }
445
446 ////////////////////////////////////
447       //
448 //this function allows showing the messages on the display
449 void SaveClass::StateDisplay(){
450
451     switch (DispStateValue) {
452
453           case DispState_A0:  if (millis() > TimeDisp + Delay){
454                                 message("   FUERA DE   ","   SERVICIO    ");
455                                 TimeDisp = millis();
456                                 }
457           break;
458
459           case DispState_A1:  if (millis() > TimeDisp + Delay){
460                                 message("   BIENVENIDO   ","       GMH      ");
461                                 TimeDisp = millis();
462                                 DispStateValue = DispState_A2;
463                                 }
464           break;
465
466           case DispState_A2:  if (millis() > TimeDisp + Delay){
467                               message("INICIE SESION", "PASE LA TARJETA");
468                               TimeDisp = millis();
469                               DispStateValue = DispState_A1;
470                                 }
471           break;
472
473           case DispState_B1: if (millis() > TimeDisp + Delay){
474                               message("   CONECTAR EL   ","     VEHICULO    ");
475                               TimeDisp = millis();
476                               }
477           break;
478
479           /* case DispState_C1: if (millis() > TimeDisp + Delay){
480                              message("VEHICULO LISTO","PARA CARGAR");
481                              TimeDisp = millis();
482                              DispStateValue = DispState_C2;
483                              }
484
485           break;
```

```
486
487             case DispState_C2: if (millis() > TimeDisp + Delay){
488                                  message("PASE UNA TARJETA","VALIDA POR FAVOR");
489                                  TimeDisp = millis();
490                                  DispStateValue = DispState_C1;
491                                }
492
493           break; */
494
495           case DispState_D1:  if (millis() > TimeDisp + Delay){
496                               message("    TARJETA    ","       LEIDA      ");
497                               TimeDisp = millis();
498                               DispStateValue = DispState_D2;
499                               }
500           break;
501
502           case DispState_D2:  if (millis() > TimeDisp + Delay){
503                               message("PROCESANDO LA","SOLICITUD AGUARDE");
504                               TimeDisp = millis();
505                               DispStateValue = DispState_D1;
506                               }
507           break;
508
509           case DispState_E1: if (millis() > TimeDisp + Delay){
510                               message("TARJETA RECHAZADA","PASE OTRA TARJETA");
511                               TimeDisp = millis();
512                               }
513           break;
514
515           case DispState_F1: if (millis() > TimeDisp + Delay){
516                               message("     VEHICULO","     CARGANDO");
517                               TimeDisp = millis();
518                               DispStateValue = DispState_F2;
519                               }
520           break;
521
522           case DispState_F2:  if (millis() > TimeDisp + Delay){
523                               message("  NO DESCONECTAR","  EL VEHICULO");
524                               TimeDisp = millis();
525                               DispStateValue = DispState_F3;
526                               }
527           break;
528
529           case DispState_F3:  if (millis() > TimeDisp + Delay){
530                               message("PASE LA TAJETA","PARA DETENER");
531                               TimeDisp = millis();
532                               DispStateValue = DispState_F1;
533                               }
534           break;
535
536           case DispState_G1: if (millis() > TimeDisp + Delay){
537                               message("CARGA FINALIZADA","DESC EL VEHICULO");
538                               TimeDisp = millis();
539                               }
540           break;
541
```

```
542           case DispState_H1: if (millis() > TimeDisp + Delay){
543                              message("FUERA DE","SERVICIO");
544                              TimeDisp = millis();
545                              }
546           break;
547
548           case DispState_I1:  if (millis() > TimeDisp + Delay){
549                              message("ERROR EN","CONECTOR");
550                                TimeDisp = millis();
551                                DispStateValue = DispState_I2;
552                              }
553           break;
554
555           case DispState_I2:  if (millis() > TimeDisp + Delay){
556                              message("    CONETOR NO ","      VALIDO");
557                                TimeDisp = millis();
558                                DispStateValue = DispState_I1;
559                              }
560           break;
561
562           case DispState_J1:  if (millis() > TimeDisp + Delay){
563                              message("CARGA RECHAZADA","DESC EL VEHICULO");
564                                TimeDisp = millis();
565                              }
566           break;
567     }
568
569 }
570
571 ////////////////////////////////////
572
573 // this function writes the messages on the display
574 void SaveClass::message(String FirstMessage, String SecondMessage){
575
576   // if (millis() > Time + Delay){
577       lcd.clear();
578       lcd.setCursor(0,0);
579       lcd.print(FirstMessage);
580       // Serial.println(FirstMessage);
581       lcd.setCursor (0,1);
582       lcd.print(SecondMessage);
583       // Serial.println(SecondMessage);
584       //Time = millis();
585   // }
586
587 }
588
589 ////////////////////////////////
590 SaveClass Save=SaveClass();
591
592 ////////////////////////////////
```

MIEM
MINISTERIO DE INDUSTRIA, ENERGÍA Y MINERÍA

MIEM - Dirección General de Secretaria
Paysandú 1101, 4° piso - C.P. 11100
Tel./Fax: (598) 2900 0231 / 33 - Fax: (598) 2900 0291
www.miem.gub.uy
Montevideo - Uruguay

RÍA DE ESTADO
ASE CITAR
19

MINISTERIO DE INDUSTRIA, ENERGÍA Y MINERÍA
MINISTERIO DE VIVIENDA, ORDENAMIENTO TERRITORIAL Y MEDIO AMBIENTE

Montevideo, 27 ENE 2020

VISTO: la necesidad de continuar con el desarrollo de la reglamentación sobre la generación de energía eléctrica sin inyección a la red del Distribuidor;----------

RESULTANDO: I) que el Reglamento de Distribución de Energía Eléctrica, aprobado por el Decreto N° 277/002 de 28 de junio de 2002, establece que las instalaciones calificadas de distribución son aquellas en Media y Baja Tensión;----------

II) que en la Sección III de dicho Reglamento no se contempló de forma explícita la generación conectada a la red de Baja Tensión, ni el uso de sistemas de acumulación (baterías) en las instalaciones de los suscritores;----------

III) que el Decreto N° 173/010 de 1° de junio de 2010 autorizó a los suscritores conectados a la red de distribución de baja tensión a instalar generación de origen renovable eólica, solar, biomasa o mini hidráulica, y a intercambiar energía en forma bidireccional con la red de Distribución;----------

IV) que el Decreto N° 43/015 de 2 de febrero de 2015, reglamentó la instalación y operación de centrales generadoras que funcionen en paralelo con la Red de Interconexión sin inyectar energía eléctrica, y las no conectadas a dicha red;----------

V) que el uso de baterías, en determinadas condiciones, puede ayudar a un mayor aprovechamiento del sistema eléctrico;----------

CONSIDERANDO: I) que en el marco de los lineamientos estratégicos de política para el sector de energía se considera conveniente diversificar la generación de energía eléctrica;----------

II) que de acuerdo a la reglamentación queda comprendido en la calidad de suscritor el titular de un suministro efectuado y medido por el Distribuidor que genera energía eléctrica para su propio consumo sin entregar energía a la red;----------

III) que se valora positivamente autorizar la generación de energía eléctrica con baterías sin inyección a la red del Distribuidor;----------

IV) que es necesario desarrollar estudios del impacto relacionado a la instalación de baterías en el sistema eléctrico;----------

ATENTO: a lo expuesto, y a lo dispuesto en Decreto Ley N° 14.694 del 1º de setiembre de 1977, en la Ley N° 16.832 de 17 de junio de 1997, y en los Decretos N° 276/002 de 28 de junio de 2002, 277/002 de 28 de junio de 2002 y 43/015 de 2 de febrero de 2015;----------

EL PRESIDENTE DE LA REPÚBLICA

D E C R E T A :

Artículo 1º.- Autorízase a los Suscritores conectados a la Red de Distribución de Baja Tensión, a generar energía eléctrica a partir de una instalación de baterías que opere en paralelo y que no inyecten energía a la red del Distribuidor.----------

Artículo 2º.- La generación de energía eléctrica por parte de Suscritores conectados a Media Tensión, a partir de una instalación de baterías que opere en paralelo a la Red de Interconexión se regirá por el artículo 12 BIS del Decreto N° 276/002 de 28 de junio de 2002, en la redacción dada por el Decreto N° 43/015 de 2 de febrero de 2015.----------

Artículo 3º.- La generación de energía eléctrica a partir de una instalación de baterías no conectada a la Red de Interconexión se regirá por el artículo 12 del Decreto N° 276/002 de 28 de junio de 2002, en la redacción dada por el Decreto N° 43/015 de 2 de febrero de 2015.----------

MIEM
MINISTERIO DE INDUSTRIA, ENERGÍA Y MINERÍA

MIEM - Dirección General de Secretaria
Paysandú 1101, 4° piso - C.P. 11100
Tel./Fax: (598) 2900 0231 / 33 - Fax: (598) 2900 0291
www.miem.gub.uy
Montevideo - Uruguay

Artículo 4º.- Los Suscritores deberán cumplir con las condiciones técnicas específicas y suscribir previamente los convenios respectivos con el Distribuidor.----------

Artículo 5º.- Las condiciones técnicas específicas serán elaboradas por la Administración Nacional de Usinas y Trasmisiones Eléctricas (UTE) y aprobadas por la Unidad Reguladora de Servicios de Energía y Agua (URSEA), con la participación en el procedimiento respectivo del Ministerio de Industria, Energía y Minería (MIEM).----------

Los términos de los convenios serán establecidos por la Administración Nacional de Usinas y Trasmisiones Eléctricas previa opinión de la Unidad Reguladora de Servicios de Energía y Agua.----------

Artículo 6º.- El Suscritor, entre otros que correspondan, pagará los costos:

I. que insuman las modificaciones razonablemente necesarias de la red eléctrica;
II. de las instalaciones interiores para la conexión a la red;
III. del eventual y razonable acondicionamiento del puesto de medida y conexión.

Artículo 7º.- El desarrollo de la actividad de generación de energía deberá cumplir con la normativa ambiental relativa a la instalación y la disposición final de baterías.----------

Artículo 8º.- Un Suscritor no podrá estar amparado simultáneamente, por el mismo suministro, en el presente decreto y el Decreto N° 173/010 de 1° de junio de 2010.----------

Artículo 9º.- La Administración Nacional de Usinas y Trasmisiones Eléctricas, la Unidad Reguladora de Servicios de Energía y Agua y el Ministerio de Industria, Energía y Minería realizarán una evaluación del impacto de la instalación de baterías en el sistema eléctrico, incluida la pertinencia de la creación de una nueva categoría tarifaria, cuando se

encuentren instalados 10 MW de potencia instalada o se cumplan tres años de la fecha de aprobación del presente Decreto.----------

Artículo 10º.- Comuníquese, publíquese, etc.----------

Dr. TABARÉ VÁZQUEZ
Presidente de la República
Periodo 2015 - 2020

//tevideo, 13 de febrero de 2020.-

R 20.-258

VISTO el expediente EX18008998, elevado por la Gerencia Comercial, en el que la Gerencia Estudios y Procesos Comerciales eleva una nueva versión del Capítulo XXIX "Instalaciones de Autoconsumo" (IAC) del Reglamento de Baja Tensión, así como del "Convenio de Conexión para suscriptor que cuenta con una instalación de autoconsumo, generación/acumulación en baja tensión"; ----------

RESULTANDO Que por R 19.-1109 de 09-05-19 se aprobó la propuesta de lineamientos elevada por la Gerencia Comercial para el tratamiento de baterías como acumuladores de energía eléctrica en las instalaciones interiores de los clientes, a ser recogida en el Reglamento de Baja Tensión; y ----------

CONSIDERANDO: I) que del informe de 10-10-19 de la Gerencia Estudios y Procesos Comerciales surge: ----------

a) Un grupo conformado por las Gerencias Distribución y Comercial, en coordinación con la Gerencia Planificación del Abastecimiento y Medio Ambiente, elaboró la propuesta de Capítulo XXIX -Instalaciones de Autoconsumo - constituidas por una instalación Acumuladora y/o una instalación Generadora, a incorporar en el Reglamento de Baja Tensión así como el texto del Convenio de Conexión respectivo. Dicha propuesta fue analizada por representantes de la DNE y URSEA y se recogieron los comentarios y observaciones formuladas; ----------

b) Se presentan además los siguientes documentos que serán necesarios para la gestión de las instalaciones: ----------

- **i)** Capítulo XXIX "Instalaciones de Autoconsumo" (IAC) del Reglamento de Baja Tensión (fs. 27 a fs. 67); --------
- **ii)** Convenio de Conexión para suscriptor que cuenta con una instalación para autoconsumo, generación/acumulación en baja tensión (fs. 68 a 81); --
- **iii)** Formulario de solicitud de conexión de la IAC (fs. 82 a fs. 83); ----------

iv) Documento de Asunción de Responsabilidad para suscriptores que incorporen generación/acumulación (fs. 84); ----------

v) Declaración Jurada de cumplimiento con los requisitos técnicos de la instalación de generación/acumulación (fs. 85); ----------

vi) Acta de Habilitación para entrar en servicio la instalación de generación/acumulación (fs. 86); ----------

vii) Solicitud de Habilitación para entrar en servicio la instalación de generación/acumulación (fs. 87); ---------

III) que en los aspectos específicos que se stán considerando, por Decreto de 27-01-20 se dispuso: -------

Artículo 1°: Autorízase a los Suscritores conectados a la Red de Distribución de Baja Tensión, a generar energía eléctrica a partir de una instalación de baterías que opere en paralelo y que no inyecten energía a la red del Distribuidor. ----------

Artículo 4°: Los Suscritores deberán cumplir con las condiciones técnicas específicas y suscribir previamente los convenios respectivos con el Distribuidor. ----------

Artículo 5°: Las condiciones técnicas específicas serán elaboradas por UTE y aprobadas por la Unidad Reguladora de Servicios de Energía y Agua (URSEA), con la participación en el procedimiento respectivo del Ministerio de Industria, Energía y Minería (MIEM). ----------

Los términos de los convenios serán establecidos por UTE previa opinión de la URSEA. ----------

L DIRECTORIO DE U.T.E. RESUELVE: ----------

1°.- Aprobar los documentos que se indican en el literal b) del ONSIDERANDO I de esta Resolución. ----------

2°.- Oficiar a la Unidad Reguladora de Servicios de Energía y gua (URSEA) y al Ministerio de Industria, Energía y Minería MIEM) conforme lo previsto en el Decreto del Poder Ejecutivo e 27-01-20. ----------

Cúmplase por Secretaría General los oficios dispuestos; cumplido, pase a la Gerencia Comercial. ----------
RE025820 EX18008998 REP.SEC.006.20D
NO85461 (URSEA) NO85462 (MIEM)

Dr. Ing. Gonzalo Casaravilla
Presidente

Dr. Jorge J. Fachola
Secretario General

Administración Nacional de Usinas y Trasmisiones Eléctricas

Montevideo, 16 de febrero de 2020.

Sr. Presidente de la Unidad Reguladora de Servicios de Energía y Agua
Ing. César FALCÓN.

Nota N° 85461

De nuestra mayor consideración:

En el marco de lo establecido en el artículo 5° del Decreto dictado por el Sr. Presidente de la República el 27-01-10, sometemos a aprobación de esa Unidad Reguladora una nueva versión del Capítulo XXIX "Instalaciones de Autoconsumo" del Reglamento de Baja Tensión, cuyo texto acompañamos.

A su vez, como también se establece en la citada norma, en forma previa a adoptar decisión recabamos vuestra opinión respecto al proyectado "Convenio de Conexión para suscriptor que cuenta con una instalación de autoconsumo, generación/acumulación en baja tensión", que remitimos.

Sin otro particular, saludan a Ud. muy atentamente,
RE025820
EX18008998

Dr. Ing. Gonzalo Casaravilla
Presidente

Dr. Jorge J. Fachola
Secretario General

NO85461-1/1

BIBLIOGRAFÍA

[1] Bernardo Wolloch. "Llegaron los eléctricos" Semanario Búsqueda. No.1974 - del 21 al 27 de junio de 2018. https://www.busqueda.com.uy/nota/llegaron-los-electricos

[2] "Subsidios de Gobierno e Intendencia posibilitan que 54 taxis eléctricos funcionen en Montevide" portal de la Presidencia de la República Oriental del Uruguay, publicado el 19.12.2018.

https://www.presidencia.gub.uy/comunicacion/comunicacionnoticias/54-taxis-electricos-miem-ute-montevideo

[3] "Preparan llamado para ómnibus eléctricos para Montevideo" El Observador, abril de 2019.

https://www.elobservador.com.uy/nota/omnibus-electricos-para-montevideo-preparan-llamado-para-el-primer-semestre--2019485035

[4] "Carga de vehículos", Movilidad Eléctrica, https://movilidad.ute.com.uy/carga.html

[5] "Open Charge Alliance", https://www.openchargealliance.org/downloads/

[6] Web Application Messaging Protocol, https://wamp-proto.org/

[7] INTERNATIONAL ELECTROTECHNICAL COMMISSION, et al. IEC 61851-1, "Electric Vehicle Conductive Charging System-Part 1: General Requirements ", FEB. 2017.

[8] SAE INTERNATIONAL, SAE J1772, "SAE Electric Vehicle and Plug in Hybrid Electric Vehicle Conductive Coupler", OCT. 2017.

[9] "Arduino: Tecnología para todos"

https://arduinodhtics.weebly.com/iquestqueacute-es.html

[10] "Cómo funciona el Puerto Serie y la UART"

https://www.rinconingenieril.es/funciona-puerto-serie-la-uart/

[11] "Entorno de desarrollo de Arduino"

https://programarfacil.com/podcast/28-entorno-de-desarrollo-de-arduino/

[12] Joint Science Academies. (2005). Joint Science Academies' Statement: Global Response to Climate Change.

https://sites.nationalacademies.org/cs/groups/internationalsite/documents/webpage/international_080877.pdf

[13] "Scientific Consensus: Earth's Climate is Warming"

https://climate.nasa.gov/scientific-consensus/

[14] "What is the Paris Agreement?"

https://cop23.unfccc.int/process-and-meetings/the-paris-agreement/what-is-the-paris-agreement

[15] "Cuáles son los dos países que están poniendo fecha de caducidad a los automóviles que funcionan con gasolina y diésel", https://www.bbc.com/mundo/noticias-40727910

[16] "Día 2, Sesión 6: Presentación de Fernando Paganini"

https://www.youtube.com/watch?v=cEiSC_19mHY&t=221s

[17] INTERNATIONAL ORGANIZATION for STANDARDIZATION, ISO/IEC 15118, "Road vehicles — Vehicle to grid communication interface", from 2013 to 2020.

[18] Kisacikoglu, Mithat Can. "Vehicle-to-grid (V2G) reactive power operation analysis of the EV/PHEV bidirectional battery charger."(2013).

[19] Vadi, Seyfettin, et al. "A Review on Communication Standards and Charging Topologies of V2G and V2H Operation Strategies.."Energies 12.19 (2019): 3748.

[20] Khaligh, Alireza, and Michael D'Antonio. "Global trends in high-power on-board chargers for electric vehicles."IEEE Transactions on Vehicular Technology 68.4 (2019): 3306-3324.

[21] Mültin, Marc. "ISO 15118 as the Enabler of Vehicle-to-Grid Applications."2018 International Conference of Electrical and Electronic Technologies for Automotive. IEEE, 2018.

[22] Mouli, Gautham Ram Chandra, et al. "Implementation of dynamic charging and V2G using Chademo and CCS/Combo DC charging standard."2016 IEEE Transportation Electrification Conference and Expo (ITEC). IEEE, 2016.

[23] Shin, Minho, et al. "Building an interoperability test system for electric vehicle chargers based on iso/iec 15118 and iec 61850 standards". Applied Sciences 6.6 (2016): 165.

[24] Lewandowski, Christian, et al. "Interference analyses of Electric Vehicle charging using PLC on the Control Pilot."2012 IEEE International Symposium on Power Line Communications and Its Applications. IEEE, 2012.

[25] Izumi, Tatsuya, et al. "Bidirectional Charging Unit for Vehicle-to-X (V2X) Power Flow."Sei Technical Review 79 (2014).

[26] Ruthe, Sebastian, et al. "Study on V2G Protocols against the Background of Demand Side Management."Ibis 11 (2011): 33-44.

[27] F. Un-Noor, P. Sanjeevikumar, L. Mihet-Popa, and E. Hossain, "A comprehensive study of key electric vehicle (EV) components, technologies, challenges, impacts, and future direction of development," Energies, vol. 10, no. 8, Aug. 2017, Art. no. 1217.

[28] M. Jafari, A. Gauchia, S. Zhao, K. Zhang, and L. Gauchia, "Electric vehicle battery cycle aging evaluation in real-world daily driving and vehicle-to-grid services," IEEE Trans. Transp. Electrific., vol. 4, no. 1, pp. 122–134, Mar. 2018.

[29] V. Monteiro, J. G. Pinto, and J. L. Afonso, "Operation modes for the electric vehicle in smart grids and smart homes: Present and proposed modes," IEEE Trans. Veh. Technol., vol. 65, no. 3, pp. 1007–1020, Mar. 2016.

[30] M. C. Kisacikoglu, M. Kesler, and L. M. Tolbert, "Single-phase onboard bidirectional PEV charger for V2G reactive power operation," IEEE Trans. Smart Grid, vol. 6, no. 2, pp. 767–775, Mar. 2015.

[31] M. Restrepo, J. Morris, M. Kazerani, and C. A. Ca˜nizares, "Modeling and testing of a bidirectional smart charger for distribution system EV integration," IEEE Trans. Smart Grid, vol. 9, no. 1, pp. 152–162, Jan. 2018.

[32] A. O. David and I. Al-Anbagi, "EVs for frequency regulation: Cost benefit analysis in a smart grid environment," IET Elect. Syst. Transp., vol. 7, no. 4, pp. 310–317, 2017.

[33] G. Yang, E. Draugedalen, T. Sorsdahl, H. Liu, and R. Lindseth, "Design of high efficiency high power density 10.5kW three phase on-boardcharger for electric/hybrid vehicles," in Proc. PCIM Eur.; Int. Exhib. Conf. Power Electron., Intell. Motion, Renewable Energy Energy Manage., Nuremberg, Germany, 2016, pp. 1–7.

[34] J. Schmenger, S. Endres, S. Zeltner, and M. Marz, "A 22 kW on-board charger for automotive applications based on a modular design," in Proc. IEEE Conf. Energy Convers., Johor Bahru, Malaysia, 2014, pp. 1–6.

[35] J. Lu et al., "A modular-designed three-phase high-efficiency highpower- density EV battery charger using dual/triple-phase-shift control," IEEE Trans. Power Electron.., vol. 33, no. 9, pp. 8091–8100, Sep. 2018

[36] X. Wang, C. Jiang, B. Lei, H. Teng, H. K. Bai, and J. L. Kirtley, "Power-loss analysis and efficiency maximization of a silicon-carbide mosfet-based three-phase 10-kW bidirectional EV charger using variable-DC-bus control," IEEE J. Emerg. Sel. Topics Power Electron., vol. 4, no. 3, pp. 880–892, Sep. 2016.

[37] P. M. Johnson and K. H. Bai, "A dual-DSP controlled SiC MOSFET based 96 %-efficiency 20kW EV on-board battery charger using LLC resonance technology," in Proc. IEEE Symp. Ser. Comput. Intell., Honolulu, HI, USA, 2017, pp. 1–5.

[38] D. Varajao, L. M. Miranda, and R. E. Araujo, "AC/DC converter with three to single phase matrix converter, full-bridge AC/DC converter and HF transformer," U.S. Patent WO2 016 024 223, Feb. 18, 2016.

[39] F. Jauch and J. Biela, "Modelling and ZVS control of an isolated threephase bidirectional AC-DC converter," in Proc. 15th Eur. Conf. Power Electron. Appl., Lille, France, 2013, pp. 1–11.

[40] I. Subotic, N. Bodo, E. Levi, B. Dumnic, D. Milicevic, and V. Katic, "Overview of fast on-board integrated battery chargers for electric vehicles based on multiphase machines and power electronics," IET Elect. Power Appl., vol. 10, no. 3, pp. 217–229, Mar. 2016.

[41] C. Shi, Y. Tang, and A. Khaligh, "A three-phase integrated onboard charger for plug-in electric vehicles," IEEE Trans. Power Electron., vol. 33, no. 6, pp. 4716–4725, Jun. 2018.

[42] "Vision Mission", https://www.charinev.org/about-us/vision-mission/

[43] "Power Line Communication", https://www.cypress.com/applications/power-line-communication

[44] "Instalaciones de Microgeneración conectadas a la red de Baja Tensión de UTE"

https://portal.ute.com.uy/sites/default/files/generico/Cap%C3%ADtulo%20XXVIII%202020.pdf

[45] "Autorización a suscriptores conectados a la red de distribución de Baja Tensión a instalar generaciones de fuentes renovables"

https://www.impo.com.uy/bases/decretos/173-2010