



UNIVERSIDAD DE

VALLADOLID

E.T.S.I. TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS ESPECÍFICAS DE TELECOMUNICACIÓN

MENCIÓN EN SISTEMAS ELÉCTRICOS

Aplicación para la obtención y procesado de datos de vehículo eléctrico

Autor:

D. Alfredo Ferreras Bustamante

Tutor:

D. David González Ortega

Valladolid, 4 de Septiembre de 2018

TÍTULO: **Aplicación para la obtención y procesado de datos de vehículo eléctrico**
AUTOR: **D. Alfredo Ferreras Bustamante**
TUTOR: **D. David González Ortega**
DEPARTAMENTO: **Departamento de Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE: **Dña. Miriam Antón Rodríguez**
VOCAL: **D. Mario Martínez Zarzuela**
SECRETARIO: **D. David González Ortega**
SUPLENTE: **D. Francisco Javier Díaz Pernas**
SUPLENTE: **D. José Fernando Díez Higuera**

FECHA: **4 de Septiembre de 2018**
CALIFICACIÓN:

Agradecimientos

A mis padres, por apoyarme y animarme a lo largo de toda mi vida.

A mis amigos, por creer en mí durante todos estos años.

A mi tutor, David, por orientarme y guiarme en el proceso.

A todos los que han hecho que hoy sea la persona que soy.

Resumen

Este TFG explora el desarrollo de una aplicación en Python y PyQt sobre Raspberry Pi para la obtención de datos de un coche eléctrico mediante la comunicación por OBD con el CAN bus del coche, permitiendo su análisis para una conducción más eficiente. Tras la aparición y creciente presencia de los coches eléctricos en los últimos años, surge la necesidad de herramientas de diagnóstico que ayuden a procesar este tipo de información. Con este fin, se proporciona un prototipo de aplicación que con una interfaz sencilla, modular y fácilmente extensible facilite esta tarea.

Palabras clave

PyQt, OBD, Raspberry Pi, coche eléctrico.

Abstract

This bachelor thesis wades through the development of an app using PyQt running over a Raspberry Pi platform. The goal is to scan and store data from an EV using the OBD-II protocol to read the EV's CAN bus. This allows for a driving efficiency in-depth analysis. After the recent surge of EVs, new diagnostic tools are needed to keep up with current trends in the industry. Therefore, an application prototype has been conceived to fill in the gap. This software has been designed with a simple GUI and both modularity and extensibility principles in mind. The aim is to provide its future users with a valuable code-base over which new projects can be built upon.

Keywords

PyQt, OBD, Raspberry Pi, EV (electric vehicle).

Índice General

Capítulo 1. Introducción	5
1.1 Motivación y objetivos	5
1.2 Fases del proyecto.....	5
1.3 Medios y elementos empleados.....	5
1.4 Estructura de la memoria	6
Capítulo 2. Fundamentos teóricos	8
2.1 Protocolo OBD-II	8
2.1.1 Orígenes	8
2.1.2 Conector (DLC)	9
2.1.3 Engine/Electronic Control Unit (ECU)	10
2.1.4 Parámetros de identificación (PID)	11
2.1.5 Códigos de diagnóstico de problemas (DTC).....	12
Capítulo 3. Herramientas hardware y configuración	14
3.1 Raspberry Pi	14
3.2 Adafruit PiTFT Display	24
3.3 Adafruit Ultimate GPS breakout	28
Capítulo 4. Herramientas software	33
4.1 Herramientas software.....	33
4.1.1 Lenguajes de programación.....	33
4.1.2 Librerías gráficas.....	35
4.1.3 Librería Python-OBD.....	37
4.1.4 Librería GPSD.....	38
4.1.5 Protocolo NMEA.....	39
4.1.6 Librería GPS3	42
4.1.7 Simulador <i>obdsim</i>	43
4.1.8 Simulador <i>gpsfake</i>	43
4.1.9 Librería SQLite	45
4.2 Librería Qt	47
4.2.1 Modelo de objetos	47
4.2.2 Manejo sencillo de memoria	47

4.2.3	Signals y Slots	48
4.3	Criterios de selección	49
4.3.1	Lenguaje de programación	49
4.3.2	Librería para desarrollo de GUI.....	50
Capítulo 5.	Estructura del software	52
5.1	Descripción.....	52
5.2	Diálogo de bienvenida	55
5.3	Diálogo de inicio de sesión	61
5.4	Menú principal	64
5.5	Diálogo de selección de PID.....	66
5.6	Diálogo de muestra de PID	69
5.7	Ventana de procesamiento online.....	71
Capítulo 6.	Presupuesto	74
Capítulo 7.	Conclusiones y líneas futuras	75
Bibliografía		78

Índice de figuras

Figura 2.1 Ejemplo de luz MIL	8
Figura 2.2 Detalle de los pines de un conector OBD-II. Puede usarse en configuraciones de 16, 6 o 9 pines. (Barreto, 2017)	10
Figura 3.1 Vista de plano de una Raspberry Pi 2 v1.1	15
Figura 3.2 Herramienta de grabado Etcher.	16
Figura 3.3 Configuración del cliente PuTTY	19
Figura 3.4 Consola bash de raspbian tras introducir la contraseña.	19
Figura 3.5 Menú de configuración raspi-config.	20
Figura 3.6 Configuración de una nueva conexión VNC.	21
Figura 3.7 Entorno de escritorio Pixel de la RPI	22
Figura 3.8 X11 <i>forwarding</i> activado para permitir la ejecución de aplicaciones gráficas.....	23
Figura 3.9 Trabajo simultáneo con consola e interfaz gráfica sobre un PC host con Windows 10.	24
Figura 3.10 Vista frontal anterior de plano de la pantalla PiTFT resistiva de 2.8”	24
Figura 3.11 Vista frontal posterior de plano de la pantalla PiTFT resistiva de 2.8” ...	25
Figura 3.12 Selección de tipo de pantalla.....	26
Figura 3.13 Selección formato apaisado no invertido e instalación y configuración del software de la pantalla.....	27
Figura 3.14 Selección modo gráfico/consola.....	27
Figura 3.15 Pines para las conexiones (izquierda), módulo GPS y socket para pila de botón (derecha).	29
Figura 3.16 Cable USB – TTL serie de Adafruit.....	30
Figura 3.17 Conexión del cable TTL serie – USB con el GPS.	31
Figura 4.1 Ejemplo de GUI diseñada con Tkinter. Aplicación para leer y codificar archivos. (Sam, 2011)	36
Figura 4.2 Interfaz gráfica del simulador <i>obdsim</i>	43
Figura 4.3 Salida de consola del simulador <i>obdsim</i>	44
Figura 4.4 Arquitectura cliente - servidor de una base de datos relacional (RDBMS) (SQLite Tutorial, 2016).	46
Figura 4.5 Arquitectura de una base de datos SQLite (SQLite Tutorial, 2016).	46

Figura 5.1	Diagrama de clase de 'UserData'	55
Figura 5.2	Estructura del archivo <i>user_list.cfg</i>	56
Figura 5.3	Estructura del archivo <i>car_list.cfg</i>	57
Figura 5.4	Estructura del archivo <i>pid_list.cfg</i>	58
Figura 5.5	Diagrama de clase de WelcomeDialog.....	58
Figura 5.6	Diagrama de clase de LoginDialog	61
Figura 5.7	Diagrama de clase de MainMenuDialog	64
Figura 5.8	Diagrama de clase PidSelectDialog	66
Figura 5.9	Cuerpo de la función <i>next_pid</i>	68
Figura 5.10	Cuerpo de la función <i>previous_pid</i>	68
Figura 5.11	Diagrama de clase PidShowDialog.....	69
Figura 5.12	Diagrama de la clase WorkerThread.....	70
Figura 5.13	Diagrama de la clase OnlineDialog.....	71
Figura 5.14	Diagrama de la clase DialogThread.....	72

Capítulo 1. Introducción

1.1 Motivación y objetivos

A lo largo de las últimas décadas del siglo XX, la tierra ha ido calentándose peligrosamente debido a la influencia del hombre. La sociedad cada vez es más consciente del enorme perjuicio que los medios de transporte (entre otros factores) causan al medio ambiente. Los automóviles constituyen una gran parte del problema, y tecnologías no contaminantes como los automóviles eléctricos, parte de la solución.

No obstante, debido a su relativamente reciente introducción en el mercado, no existe un abanico de herramientas para diagnóstico para automóviles eléctricos tan amplio como en el caso de los automóviles de combustión tradicionales.

En este contexto, se ha planteado la necesidad de desarrollar en este TFG una aplicación para el procesado de datos de un vehículo eléctrico. El objetivo es, partiendo de tecnologías ya existentes, como Raspberry Pi, Python, PyQt o el protocolo OBD, conseguir una herramienta que facilite el diagnóstico en coches eléctricos. También se marca como objetivo la visualización de datos de la eficiencia de conducción en tiempo real.

1.2 Fases del proyecto

El desarrollo de este TFG incluye varias partes claramente diferenciadas; en primer lugar, recopilación de información para el desarrollo. En segundo lugar, discusión con el tutor de las herramientas más adecuadas. En tercer lugar, obtención y configuración de los equipos físicos. En cuarto lugar, prueba de los módulos software por partes y retroalimentación por parte del tutor. En quinto lugar, diseño e implementación por partes del sistema completo. Por último, pruebas en carretera, retroalimentación y revisión del código para plantear posibles mejoras de cara a otros futuros proyectos.

1.3 Medios y elementos empleados

En la primera fase, se emplearon artículos de investigación y bibliografía de introducción a la Raspberry Pi. En la segunda fase, se barajaron varios lenguajes como C, C++, Python y librerías gráficas como WxWidgets, Qt y PyQt. También se discutió sobre la posibilidad de adquirir hardware como el módulo GPS y de la pantalla PiTFT. Durante la tercera fase, se investigó y se probó la configuración de

todos los elementos hardware y software de manera individual, tal y como se explica en los capítulos 3 y 4. A continuación, en la fase cuatro, se hicieron pequeñas pruebas por separado de cada una de las herramientas para verificar su funcionamiento. En la fase cinco, se emplearon herramientas de programación como QtCreator (para el desarrollo de la interfaz gráfica) o Geany (para desarrollar el software en Python).

1.4 Estructura de la memoria

El capítulo 1 explica las motivaciones y objetivos del TFG, desarrolla las fases y explica los medios empleados, además de ofrecer una introducción a la estructura del TFG.

El capítulo 2 presenta una introducción somera del protocolo OBD.

En el capítulo 3 se presentan las herramientas hardware que se van a usar durante este TFG. Se exponen las razones de selección de la Raspberry Pi, y se explica cómo acceder a ella y controlarla de manera remota sin necesidad de conectar una pantalla externa por HDMI. Se explica también cómo configurar por primera vez la pantalla PiTFT, así como el módulo GPS. También se hace una descripción detallada de los scripts que permiten configurarlo correctamente y de manera sencilla.

En el capítulo 4 se introducen las herramientas software que van a utilizar durante el proyecto:

- Se realiza una comparativa entre los lenguajes de programación que podrían servir y se explica el porqué de la selección de Python, así como el resto de librerías (obd, gpssd y gps3). También se comenta brevemente el protocolo NMEA usado por todos los módulos GPS, incluyendo el de este TFG.
- Se realiza una descripción a nivel profundo de la librería Qt que da soporte a la interfaz gráfica. Se explica su modelo de objetos (cómo se construyen internamente las ventanas), cómo el manejo de memoria es más sencillo que en C++ y cómo se comunican las ventanas y objetos Qt entre sí, mediante el mecanismo de señales y ranuras.
- Se exponen las razones por las que PyQt y Python han sido seleccionados como herramientas para construir el software, y por qué las alternativas no ofrecían el mismo nivel.

En el capítulo 5 se hace un repaso muy exhaustivo de la estructura profunda del software. En él, se recorre paso a paso cada clase, explicando sus métodos y variables y cómo se han implementado las diferentes librerías y herramientas descritas en el capítulo 4. También se justifican las diferentes decisiones de diseño y se ofrece una visión bastante integradora de todas las funciones.

En el capítulo 6 se realiza un presupuesto desglosando el coste de cada elemento.

En el capítulo 7 se exponen finalmente las conclusiones y se desarrollan brevemente posibles líneas futuras de trabajo.

Capítulo 2. Fundamentos teóricos

2.1 Protocolo OBD-II

2.1.1 Orígenes

En los años 70, diversos estudios científicos comenzaron a vincular la contaminación y el deterioro del medio ambiente. Se vio especialmente la mala influencia de emisiones de productos tóxicos procedentes de los medios de transporte de masas, y en concreto, del medio de transporte por excelencia: el automóvil.

Fue entonces cuando las diferentes casas de fabricantes de automóvil pasaron a la acción y decidieron emplear dispositivos para luchar contra los subproductos nocivos de los motores de combustión. Los primeros intentos afectaban seriamente al empuje de los vehículos, y no fue hasta la invención del convertidor catalítico cuando por fin se dio el salto y comenzaron a extenderse por los modelos de coche más populares.

En EEUU, estados como California fueron pioneros en estas regulaciones y comenzaron a exigir en los nuevos vehículos la inclusión de estos incipientes sistemas de control de emisiones. Hacia 1980, cuando aún solo California los exigía en sus regulaciones, General Motors desarrolló el primer sistema de diagnóstico a bordo, más conocido como OBD. Su misión era posibilitar el diagnóstico de fallos en los sensores que registraban las emisiones de los vehículos, para así comprobar si éstos se ajustaban a la legislación vigente.

Sin embargo, y pese a ser un gran avance, las limitaciones de este sistema se hicieron patentes, pues apenas se guardaban datos sobre los niveles de gases. Además, cada fabricante empleaba sus propios DTC (*Diagnostic Trouble Codes*) y MIL (*Malfunction Light Indicators*) (Coria, 2014).



Figura 2.1 Ejemplo de luz MIL

A principios de los años 90 el congreso de los EEUU introdujo la Ley de Aire Limpio con el objetivo de estandarizar entre fabricantes y modelos el sistema de diagnóstico de a bordo, naciendo OBD-II. Esta nueva ley permitió a la EPA (Environmental Protection Agency) obligar a extender a todo el país las regulaciones de la agencia homóloga californiana (California Air Resources Board), ya vigentes desde hace tiempo en dicho estado.

Los fabricantes de automóviles adaptaron de forma inmediata sus sistemas a la nueva regulación. Hoy en día su uso también es obligatorio en la UE para todos los coches fabricados desde el año 2002. (Coria, 2014)

Hoy en día, **OBD-II (On-Board Diagnostics)** es un conjunto de especificaciones pensadas para monitorizar e informar de las emisiones, rendimiento y fallos de los motores de los automóviles.

También se utiliza para comprobar el estado general de cada una de las partes de un coche, comunicándose con los sistemas electrónicos (**ECUs**) que las gobiernan. En general, *Electronic Control Unit* se refiere a un sistema empotrado (*hardware*) que controla un subsistema del automóvil (p.ej. los frenos ABS). *Engine Control Unit* o *Engine Control Module* se refiere al ECU concreto que controla un conjunto de actuadores que garantizan un rendimiento óptimo del motor.

En la práctica, OBD es un término que abarca una familia de protocolos empleados en automoción, normalmente propietaria. Funcionan bajo una interfaz eléctrica única, activando diferentes pines según sea necesario. (Corinne, 2018.). Así, el sistema OBD está formado por el conector DLC y su correspondiente ECU.

2.1.2 Conector (DLC)

En los coches, el conector o *Data Link Connector* normalmente está situado cerca del asiento del conductor o en las proximidades del cenicero. De esta manera, resulta fácilmente accesible sin necesidad de emplear herramientas específicas.

En la figura 2.1 puede observarse el detalle de los pines de un conector.

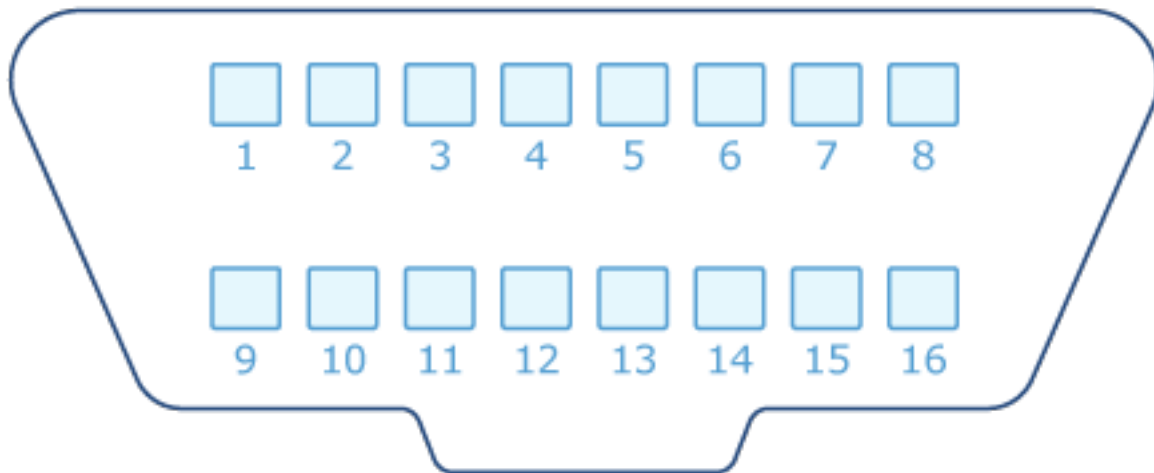


Figura 2.2 Detalle de los pines de un conector OBD-II. Puede usarse en configuraciones de 16, 6 o 9 pines. (Barreto, 2017)

2.1.3 Engine/Electronic Control Unit (ECU)

Las ECUs constituyen el cerebro del vehículo. Monitorizan y controlan muchas de las funciones del coche. Estas funciones pueden ser estándares (definidas por el fabricante) o reprogramables. También pueden conectarse y usarse en cascada para maximizar sus ventajas. En el caso de la ECU encargada del motor (ECM), el ajuste de los diferentes parámetros permite que el motor trabaje en varias configuraciones, según se desee mayor rendimiento o mayor ahorro. En automóviles nuevos, todas las ECUs suelen ser microcontroladores. (Corinne, 2018)

A continuación se enumeran algunas de las ECUs más comunes:

- **Engine Control Module (EDM)**: controla los actuadores del motor, afectando a aspectos como el tiempo de encendido, las ratios aire/combustible y velocidad de ralentí.
- **Vehicle Control Module (VCM)**: controla el rendimiento del motor y del vehículo.
- **Transmission Control Module (TCM)**: maneja la transmisión, incluyendo elementos como la temperatura del fluido de transmisión, la posición del acelerador y la velocidad de la rueda.
- **Powertrain Control Module (PCM)**: normalmente es la combinación de un ECM y un TCM. Controla el tren de potencia.
- **Electronic Brake Control Module (EBCM)**: controla y lee los datos del ABS (del alemán *Antiblockiersystem* o sistema antibloqueo de ruedas,

que dosifica la fuerza de frenado para mejorar el agarre de los neumáticos a la superficie, evitando que se bloqueen. De esta manera, el conductor conserva el control sobre la trayectoria del vehículo, pudiendo rectificarla con el volante ante la presencia de obstáculos).

- **Body Control Module (BCM):** controla las características del chasis del vehículo, tales como ventanas eléctricas, asientos eléctricos, etc. (Corinne, 2018)

2.1.4 Parámetros de identificación (PID)

Sea cual sea el protocolo utilizado, OBD presenta 10 modos de diagnóstico. No todos los modos son necesariamente compatibles con la ECU del motor (ECM). Cuanto más reciente sea el vehículo, mayor será la posibilidad de que admita más modos. La siguiente lista de vehículos compatibles con OBD2 ofrece algunos ejemplos de vehículos probados por los usuarios. (Outils OBD Facile, 2018)

Modo	Descripción
I	<p>Este modo devuelve los valores comunes para algunos sensores tales como: la velocidad del motor, la velocidad del vehículo, la temperatura del motor (aire, refrigerante) o la información sobre los sensores de oxígeno y la mezcla aire/combustible.</p> <p>Cada sensor se caracteriza por un número llamado PID (Identificador de parámetro) utilizado para identificar el parámetro. Por ejemplo, el estándar estipula que la velocidad del motor tiene un PID de 12. El estándar OBD (actualizado en 2007) incluye 137 PID. En cuanto a los modos, no todos los autos admiten todos los PID. La lista de páginas de vehículos compatibles con OBD-II proporciona los PID admitidos en los diversos modos en ciertos vehículos.</p>
II	<p>Este modo proporciona datos de un fallo en un instante. Cuando el ECM detecta un fallo, registra los datos del sensor del momento específico en el que aparece. Será el modo en el que trabajará el software de este TFG.</p>
III	<p>Este modo muestra los códigos DTC. Estos códigos de fallo son estándar para todas las marcas de vehículos y se dividen en 4 categorías:</p> <p>P0xxx: fallos estándar vinculadas al tren motriz (motor y transmisión).</p> <p>C0xxx: fallos estándar en el chasis.</p> <p>B0xxx: fallos estándar en el cuerpo.</p>

	U0xxx: fallos estándar en la red de comunicaciones.
IV	Este modo se usa para borrar los códigos de fallo grabados y apagar el indicador de fallo del motor (MIL). En general no es necesario borrar un fallo que no ha sido diagnosticado o reparado. La MIL se encenderá de nuevo durante la próxima sesión de conducción.
V	Este modo proporciona los resultados del autodiagnóstico realizado en los sensores de oxígeno. Se aplica principalmente solo a los vehículos de gasolina. Para las nuevas ECU que usan CAN, ya no se usa.
VI	Este modo da los resultados del autodiagnóstico hecho en sistemas no sujetos a vigilancia constante. El Modo 6 reemplaza las funciones que estaban disponibles en el Modo 5.
VII	Este modo proporciona códigos de fallo no confirmados. Después de una reparación es muy útil verificar que el código de fallo no vuelva a aparecer sin tener que realizar una prueba larga. Los códigos utilizados son idénticos a los del modo 3.
VIII	Este modo proporciona los resultados del autodiagnóstico en otros sistemas. Apenas se usa en Europa.
IX	Este modo proporciona la información relativa al vehículo, como por ejemplo el VIN (número de identificación del vehículo) o ciertos valores de calibración.
X	También conocido como modo A. Este modo proporciona los códigos de fallo permanentes. Los códigos utilizados son idénticos a los de los modos 3 y 7. A diferencia de los modos 3 y 7, estos códigos no se pueden borrar utilizando el modo 4. Solo varias sesiones de conducción en carretera con el problema que los causó resuelto pueden borrarlos.

2.1.5 Códigos de diagnóstico de problemas (DTC)

Estos códigos sirven para identificar un problema en caso de que ocurra. Están definidos bien por la SAE (*Society of Automotive Engineers*), bien por los propios fabricantes, es decir, pueden ser genéricos o específicos de cada fabricante. A continuación puede observarse la estructura de uno de estos códigos (Corinne, 2018):

1 2 3 4 5

El primer dígito identifica el tipo de código de error:

- P____ para el tren de potencia.
- B____ para el cuerpo.
- C____ para el chasis.
- U____ para la red de clase 2.

El segundo dígito muestra si el código es único del fabricante o no:

- _0___ para códigos requeridos por el gobierno.
- _1___ para códigos requeridos por el fabricante.

El tercer dígito muestra a qué sistema se refiere el código de fallo:

- __1__ / __2__ hacen referencia a mediciones de aire y combustible.
- __3__ hace referencia al sistema de arranque.
- __4__ hace referencia al sistema de emisiones.
- __5__ hace referencia al control de velocidad/reposo.
- __6__ lidia con sistemas de ordenadores.
- __7__ / __8__ hace referencia a la transmisión.
- __9__ denota señales de entrada/salida y controles.

Los dígitos cuatro y cinco muestran el código de fallo específico.

- ____00 a ____99 dependen de los sistemas definidos en el tercer dígito.

Capítulo 3. Herramientas hardware y configuración

A continuación, se procede a describir las características y configuración de las herramientas hardware para este TFG.

3.1 Raspberry Pi

Existen una serie de motivos por los que merece la pena usar una Raspberry Pi en vez de otras alternativas:

- **Arquitectura similar a PC:** esto permite transferir de manera automática todas las habilidades y conocimientos para manejar un PC al ámbito de los sistemas embarcados. Puesto que en el fondo no es más que un PC en miniatura, el usuario puede comenzar a usar el sistema Linux de manera inmediata.

Un microcontrolador requiere un aprendizaje específico asociado a cada plataforma, y eso inevitablemente encarece el proceso de prototipado y diseño en términos de tiempo. Además, en un microcontrolador no existe un Sistema Operativo al que retornar, por lo que el manejo de periféricos es muy ad-hoc y específico de cada arquitectura. Además, aunque son perfectos para resolver cuestiones puramente electrónicas, si se desea tratar con grandes cantidades de información resulta infinitamente más cómodo trabajar con librerías ya existentes. No hay necesidad de "reinventar la rueda". Permite también usar todas las herramientas desarrolladas para plataformas de tipo ARM, como compiladores, librerías gráficas, incluso los entornos de desarrollo, facilitando enormemente en definitiva la transición.

- **Precio:** con un coste aproximado de 35 €, la Raspberry Pi se ha erigido como alternativa frente a otros sistemas embarcados menos funcionales e incluso más caros. En el fondo es un ordenador, por lo que se pueden realizar tareas como servidores, automatización o streaming de vídeo.
- **Experimentar con seguridad:** el ordenador es una herramienta trabajo diario. Un usuario, para aprender o para prototipar debe, en primer

lugar, experimentar con diferentes configuraciones e incluso disponer de la capacidad de instalar/reinstalar el SO de manera rápida o efectiva. Otras alternativas como los microcontroladores no son tan versátiles y requieren de plataformas o placas que actúen como banco de pruebas independiente, aumentando su costo.

Configuración de la Raspberry Pi

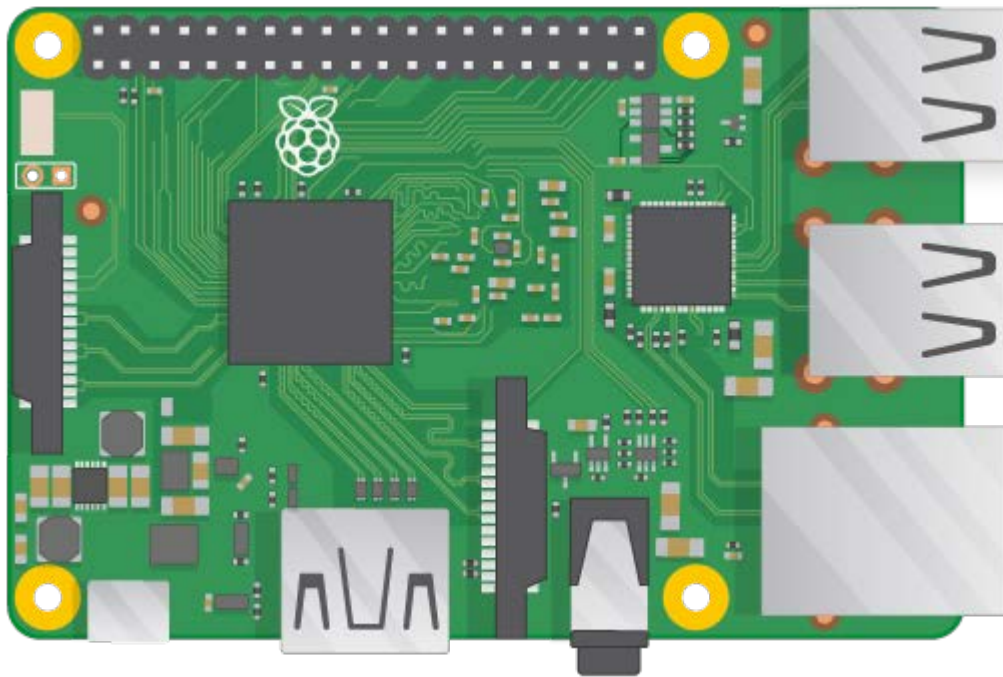


Figura 3.1 Vista de plano de una Raspberry Pi 2 v1.1

La Raspberry Pi dispone de dos formas de instalar su sistema operativo: la primera, mediante un instalador y gestor de arranque llamado NOOBS (*New Out Of the Box Software*), y la segunda mediante Etcher (un software grabador de imágenes .iso en memorias flash).

Aunque el fabricante recomienda el uso de NOOBS, el proceso de instalación es más lento, pues requiere:

1. Grabar la imagen ISO de NOOBS en la tarjeta SD.
2. Iniciar la RPI. Para poder controlarla, se necesita conectar un teclado USB, un ratón USB y una pantalla por HDMI.
3. Seleccionar la imagen de Raspbian o Raspbian Lite (sistema mínimamente funcional) y aguardar a que termine el proceso de instalación.

También están a la venta tarjetas SD con NOOBS preinstalado. Este método está enfocado primordialmente a usuarios muy noveles o poco diestros en computación.

En la práctica y para una persona mínimamente experimentada, resulta mucho más rápida la segunda opción:

1. Descargar la imagen ISO de Raspbian Stretch (última versión en la fecha de redacción de este TFG).
2. Grabarla mediante la herramienta Etcher. El SO es completamente funcional, sin necesidad de instalación durante el primer inicio. Se trata de un proceso análogo al procedimiento de *flashing* del software de un teléfono inteligente.

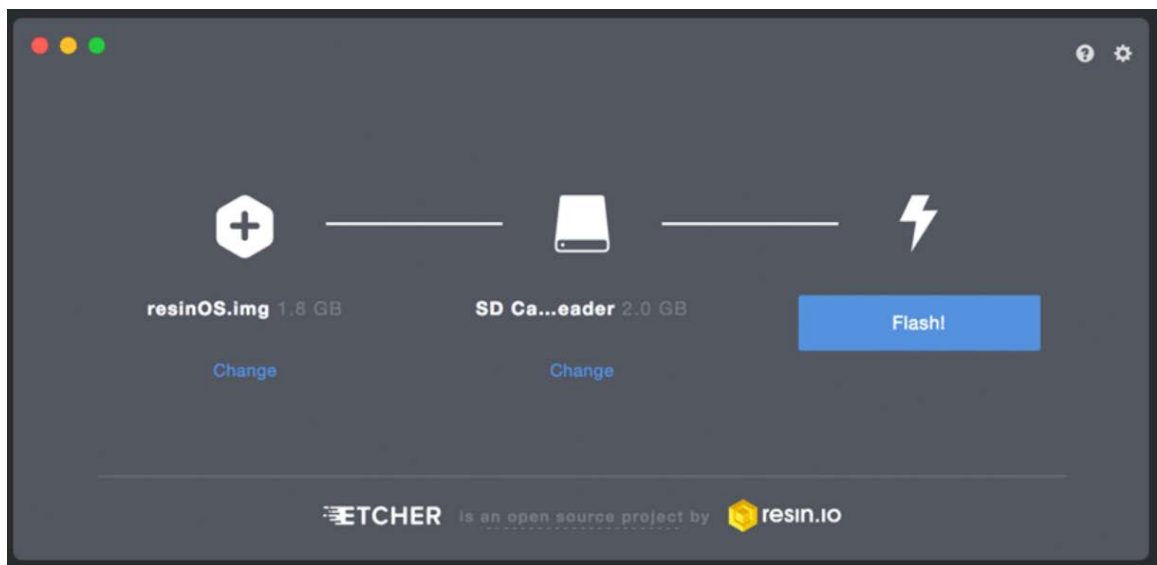


Figura 3.2 Herramienta de grabado Etcher.

3. Abrir el explorador de archivos del SO y crear un archivo vacío ssh.txt en el directorio raíz de la SD recién grabada. Con esto, la RPI activará la conexión SSH durante su primer inicio y bastará con una conexión Ethernet y un terminal SSH para comenzar a manejarla desde cualquier PC. Este modo de trabajo se conoce como *SSH headless*.

En cualquier sistema *nix bastará con abrir el terminal y escribir:

```
cd <ruta_de_raiz_SD>  
touch ssh.txt
```

Una vez iniciada la Raspberry Pi y conectada por Ethernet al PC, es necesario establecer una conexión SSH para poder acceder a la terminal bash de la RPI.

En sistemas *nix, bastará con:

```
ssh pi@<dir_IP_RPI>
```

El problema es que la dirección IP de la RPI no es conocida. Puesto que es muy habitual trabajar en una red local con direcciones IP dinámicas, el mayor inconveniente que uno se encuentra es precisamente el no saber la dirección IP asociada a un equipo en un momento determinado. Existen varias formas de solucionarlo, incluyendo fijar una IP estática en los archivos de configuración de Raspbian antes de iniciar el SO, pero la alternativa más rápida es el empleo de un sistema mDNS o *multicast DNS*.

Normalmente, todos los equipos al conectarse a una red necesitan consultar a un servidor DNS para traducir el nombre a una dirección IP concreta.

Sin embargo, esta situación también puede darse con direcciones IP locales; es decir, cuando se desconoce la IP concreta mientras que sí se conoce el nombre del equipo. La idea de mDNS es precisamente esta: en ausencia de un servidor o equipo que centralice la resolución de nombres, cada equipo tiene la misión de resolver su propio nombre al resto, mediante un sistema multicast. Existen diferentes implementaciones, siendo las más conocidas Bonjour (para macOS, compatible también con Windows) y Avahi (GNU/Linux y BSD).

Para macOS, Bonjour está instalado por defecto.

Para Windows, se puede descargar gratis de la página web de Apple.

La mayor parte de las distribuciones GNU/Linux instalan por defecto el demonio avahi (*avahi-daemon*) y el paquete *libnss-mdns*. Para que pueda utilizarse mDNS, hay que asegurarse de que el fichero `/etc/nsswitch.conf` contenga la línea:

```
hosts: files mdns4_minimal [NOTFOUND=return] dns mdns4
```

En lugar de la habitual:

```
hosts: files dns
```

De esta manera, se harán resoluciones de nombres a través de mDNS (IPv4) de manera automática, sin necesidad de conocer la dirección IP concreta y a partir del nombre del equipo.

Por defecto, la RPI tiene de nombre de equipo *raspberrypi*, de usuario *pi* y de contraseña *raspberry*. Así que, una vez configurado el sistema mDNS, en un sistema *nix bastará con:

```
ssh pi@raspberrypi.local
```

El sistema mDNS se encargará de traducir *raspberrypi.local* a una IP concreta. Se preguntará al usuario sobre si confiar en el host y guardar la clave. Se seleccionará “yes” y se escribirá la contraseña *raspberry*.

Esto nos llevará a una terminal bash de la RPI y nos permitirá controlar y administrar el sistema Raspbian instalado.

Configuración de la conexión por SSH en Windows

Tanto la ventana de comandos clásica de Windows como la Powershell carecen de capacidad de establecer una conexión SSH. Por lo tanto, resulta imprescindible usar un cliente sustituto, como PuTTY.

PuTTY es un software gratuito descargable de su página web.

Una vez instalado, basta con ejecutarlo y seleccionar SSH. Se procede entonces a rellenar los campos según corresponda y a establecer la conexión.

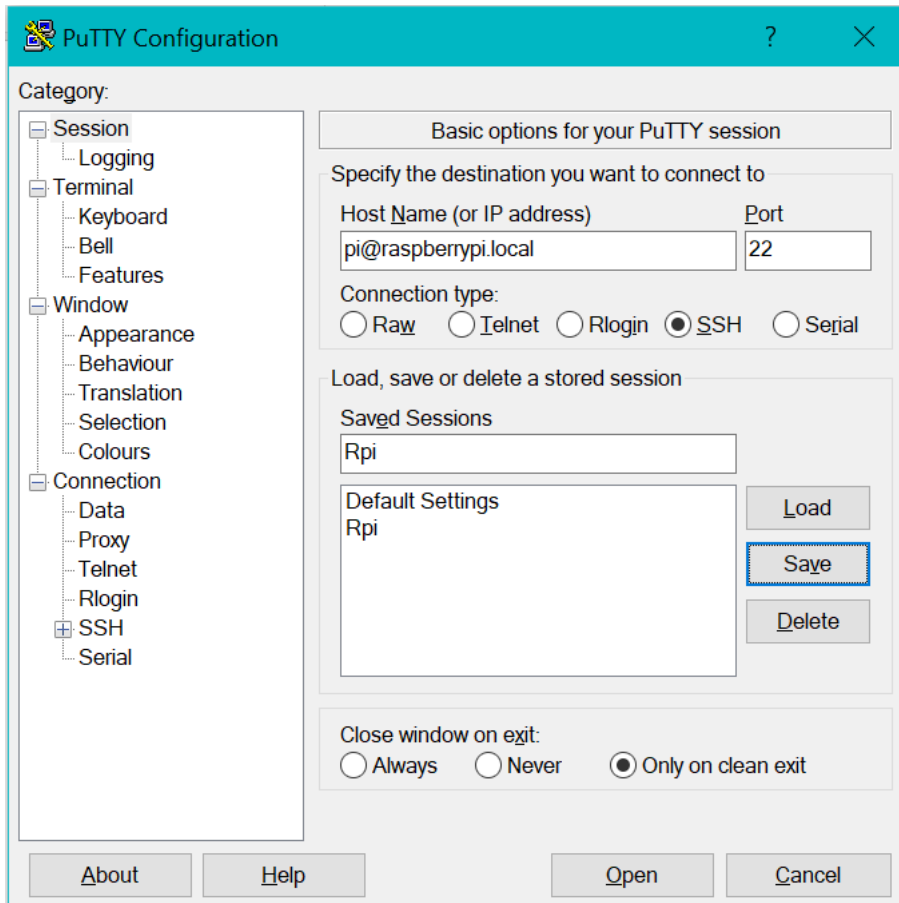


Figura 3.3 Configuración del cliente PuTTY

Pedirá la contraseña, *raspberry*, y ahora sí se podrá trabajar con una terminal de Raspbian de manera remota:

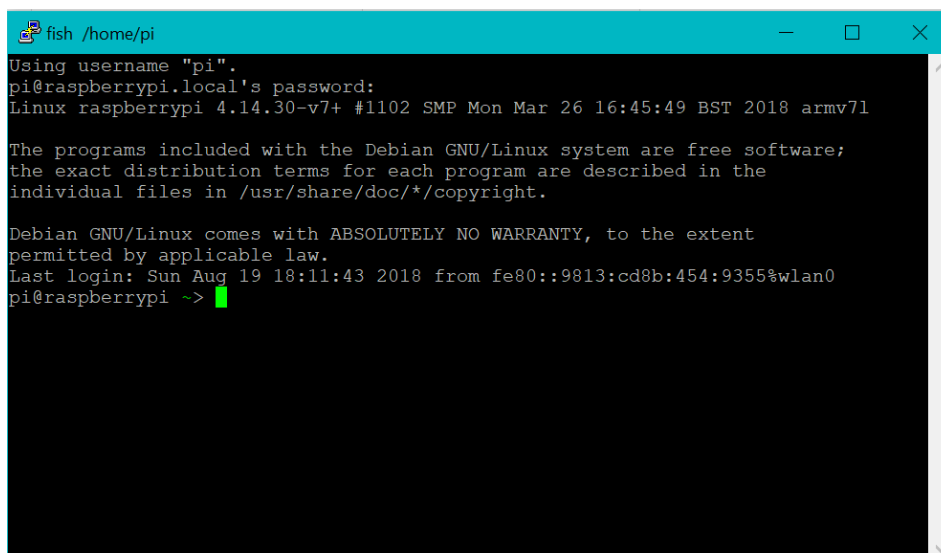


Figura 3.4 Consola bash de raspbian tras introducir la contraseña.

Configuración del sistema: *raspi-config*.

Una vez instalado el SO, basta con conectar la fuente de alimentación y el cable Ethernet al PC. Tras conectarse por SSH, se podrá cambiar la contraseña Unix por otra de elección del usuario. Para ello, basta con ejecutar:

```
sudo raspi-config
```

Aparecerá un diálogo como el de la figura 3.5:

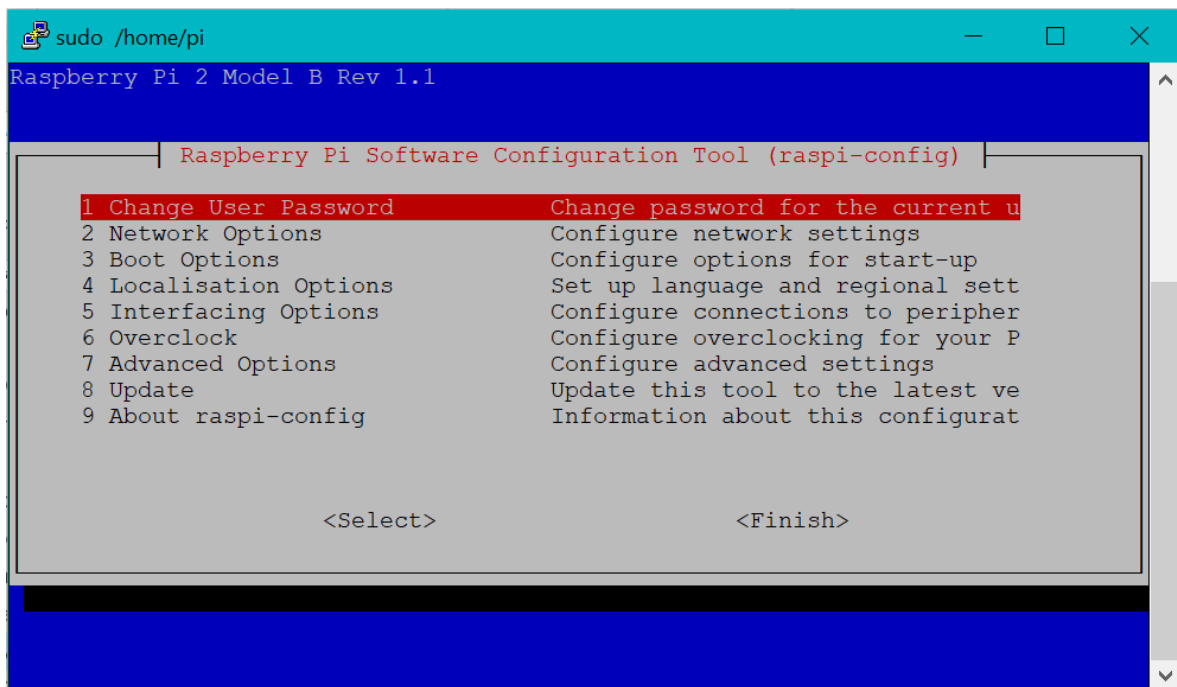


Figura 3.5 Menú de configuración raspi-config.

A partir de aquí, se pueden modificar varias opciones además de la contraseña de usuario, como el idioma o la distribución de teclado. No obstante, lo más urgente es poder tener la posibilidad de acceder gráficamente al entorno de escritorio Pixel.

Para ello, se utilizará el servidor VNC incorporado en el sistema operativo de la Raspberry. Basta con seleccionar “5 Interfacing options” y VNC <Enter>. Activaremos el servidor VNC y reiniciaremos.

VNC son las siglas de *Virtual Network Computing*, y es un protocolo muy sencillo que consta de un software cliente, otro servidor y una primitiva cuya función es transmitir pixel a pixel entre servidor y cliente la imagen del escritorio. También proporciona un canal para envío y recepción de mensajes (como Ctrl+Alt+Del).

A continuación, se seleccionará <Finish> y se aceptará reiniciar la RPI. En el PC o Mac se procederá a instalar el VNC Viewer de RealVNC

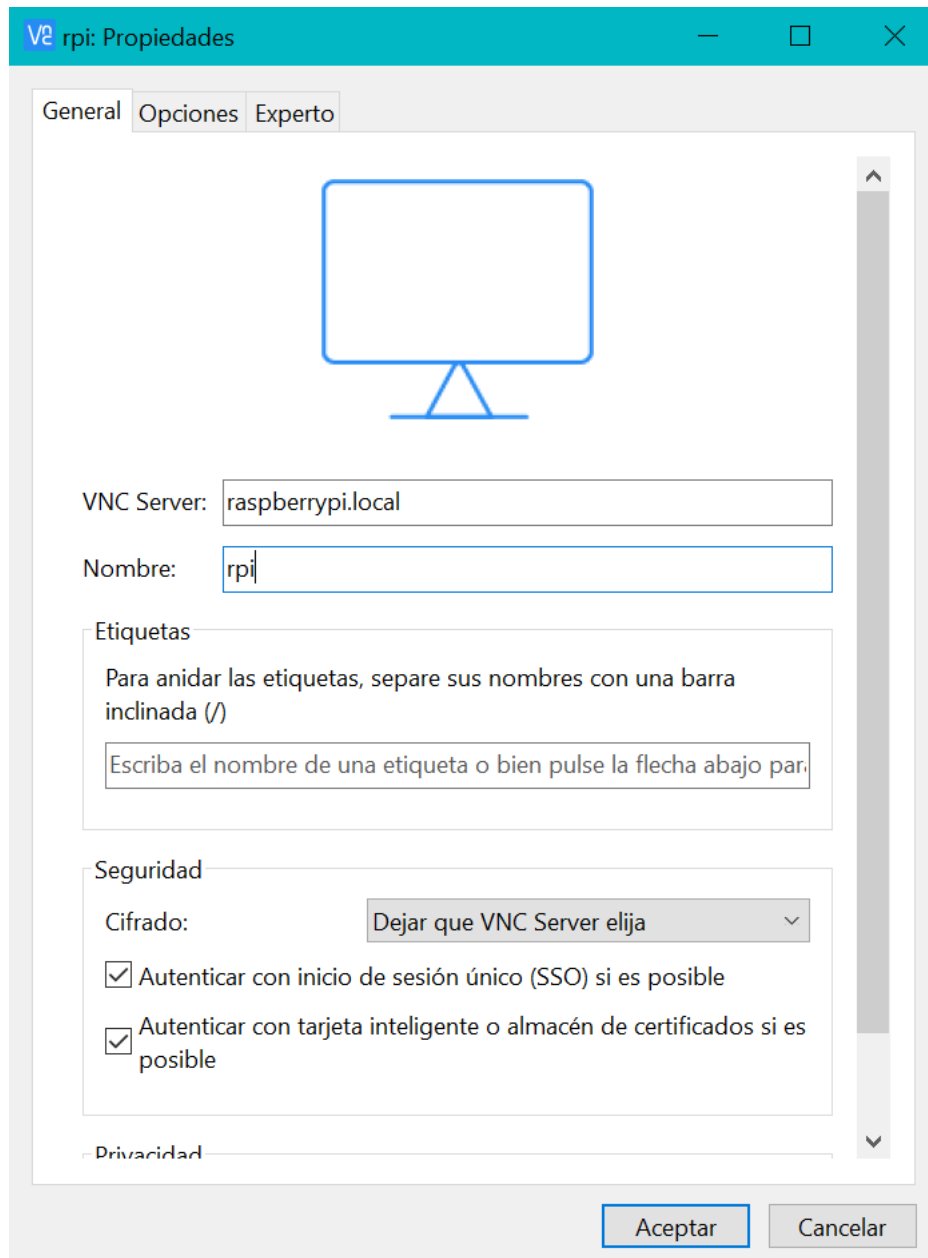


Figura 3.6 Configuración de una nueva conexión VNC.

Abriendo el visor de VNC, se procede a crear una nueva conexión y completar los datos como en la figura 3.6.

Se pulsa aceptar y bastará con hacer doble clic en el icono que se ha creado, introducir el usuario y la contraseña; por último, habrá que aceptar el mensaje que salta e inmediatamente aparecerá una ventana con el escritorio Pixel listo para utilizar.

De esta manera es posible también ejecutar aplicaciones gráficas.

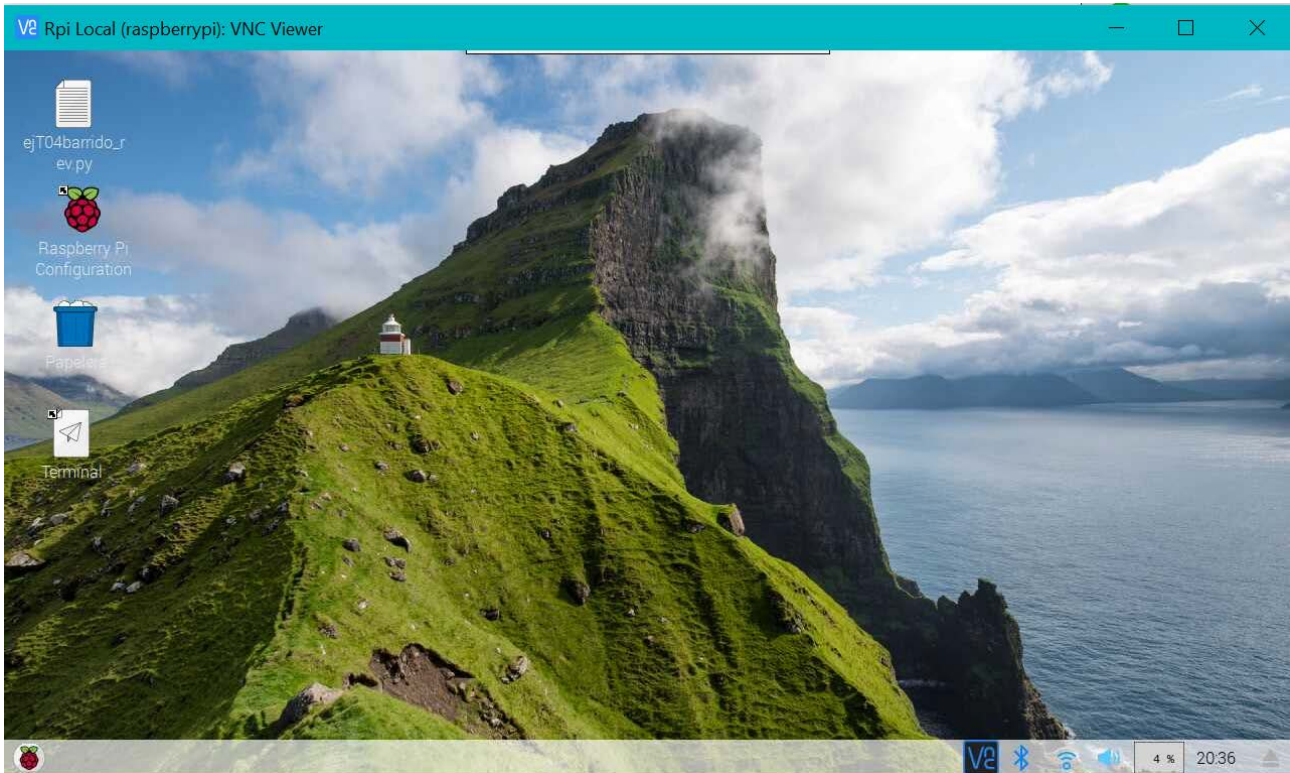


Figura 3.7 Entorno de escritorio Pixel de la RPI

Aplicaciones gráficas

Salvo que se disponga de dos pantallas, resulta incómodo cambiar continuamente entre aplicaciones gráficas y terminal SSH. Además, las ventanas gráficas tienen como límite de resolución la de la propia ventana del visor VNC. Lo ideal sería, por lo tanto, combinar ventanas de aplicaciones gráficas de la propia RPI con las ventanas del SO del PC (como el terminal SSH). Esto puede hacerse mediante el servidor X11.

Para sistemas Windows, basta con instalar el servidor Xming. Una vez instalado, se inicia en la bandeja de notificaciones de la barra del escritorio. Ha de permanecer abierto.

A continuación, se configura el cliente PuTTY para permitir sesiones gráficas X11. Para ello, basta con configurarlo tal y como se indica en la figura 3.8 (Connection > SSH > X11 > *Enable X11 forwarding*).

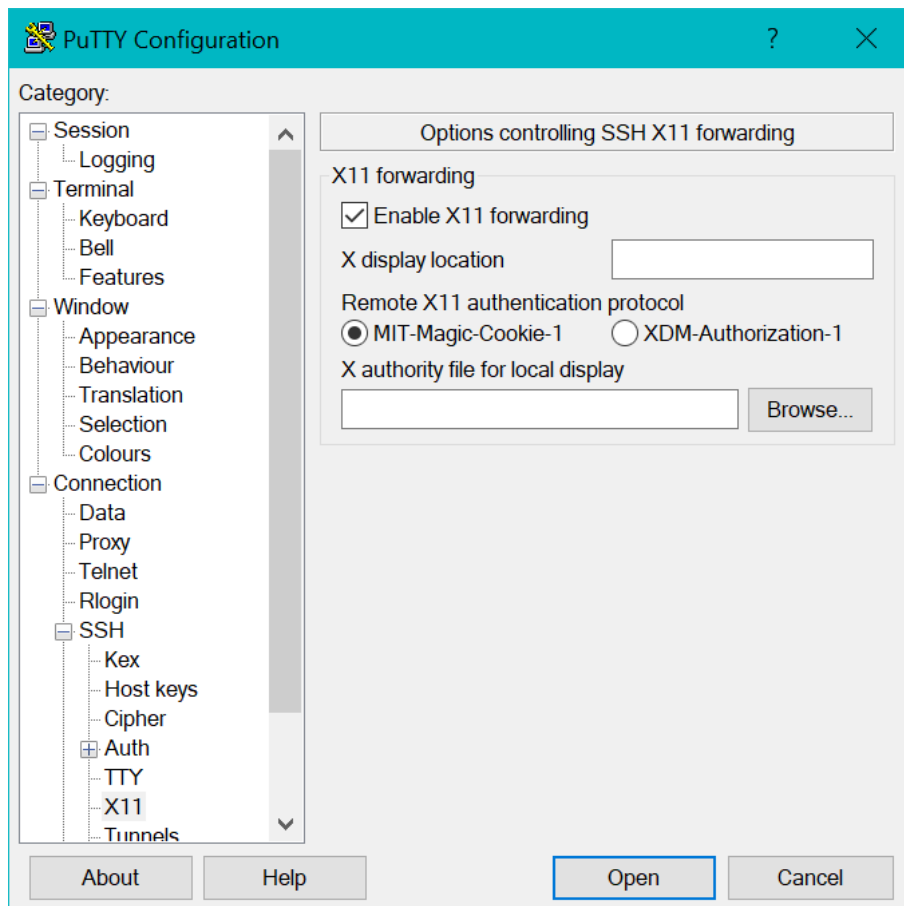


Figura 3.8 X11 *forwarding* activado para permitir la ejecución de aplicaciones gráficas.

A partir de aquí, es suficiente con iniciar una sesión SSH. Una vez dentro de la consola bash de Raspbian, será posible iniciar también aplicaciones gráficas, como `pcmanfm` (explorador de archivos), que se abrirán como ventanas del sistema operativo host (PC o MAC). De esta manera, será posible trabajar conjuntamente con varias ventanas correspondientes a aplicaciones de la RPI separadas desde un único sistema host, con el beneficio de poder ordenarlas y cambiarlas de tamaño según convenga, tal y como puede verse en la figura 3.9:

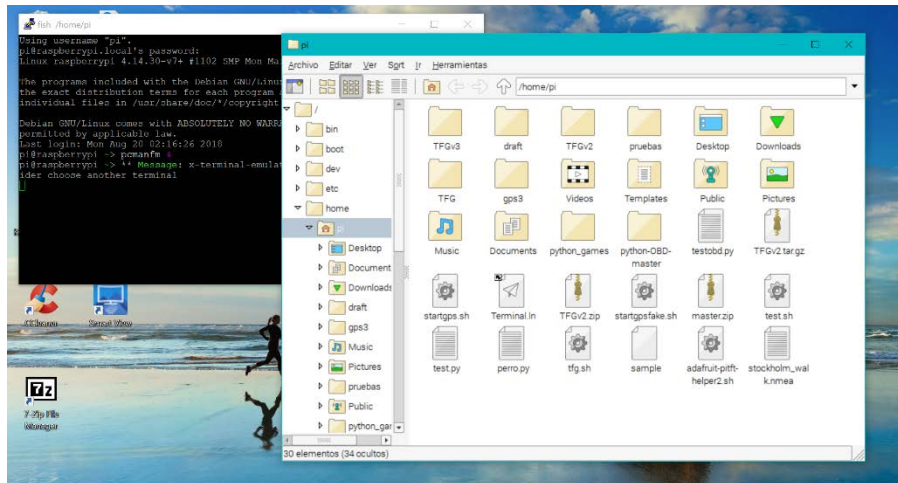


Figura 3.9 Trabajo simultáneo con consola e interfaz gráfica sobre un PC host con Windows 10.

3.2 Adafruit PiTFT Display

Para poder mostrar información básica durante la sesión de captura, y aprovechar el máximo espacio posible para visualizar elementos, se empleará una pantalla resistiva de 2.8" de Adafruit. En la figura 3.10 puede verse una vista de plano de la pantalla:

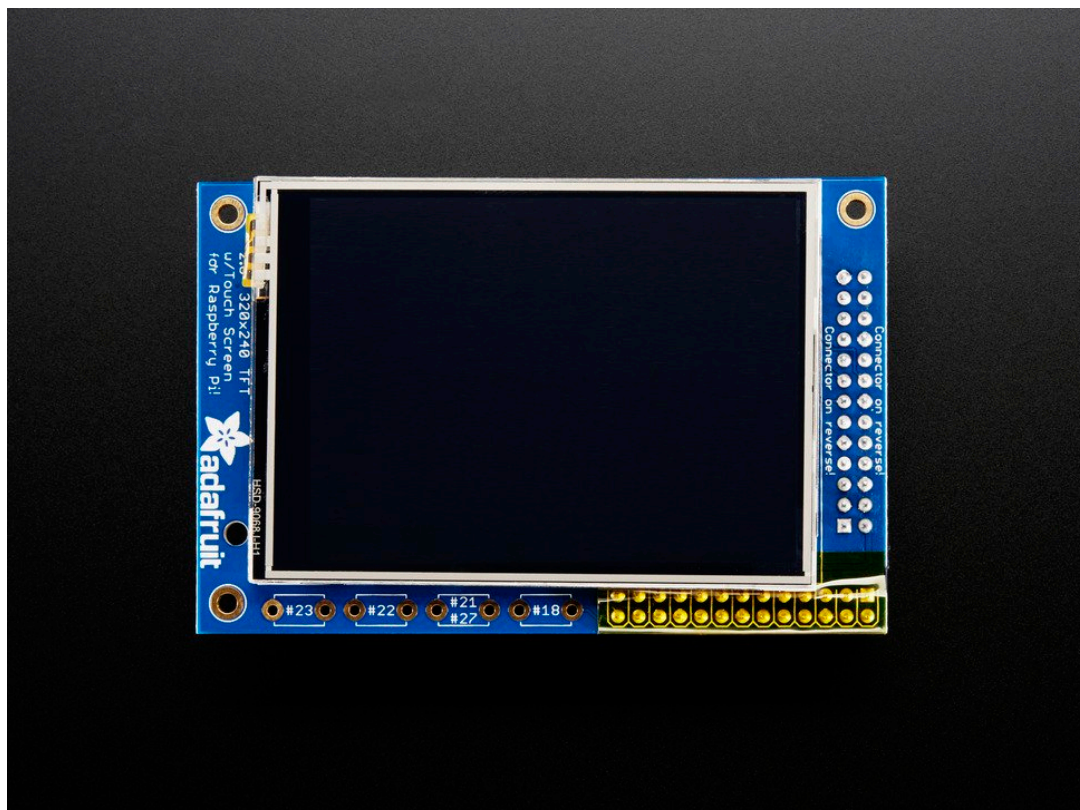


Figura 3.10 Vista frontal anterior de plano de la pantalla PiTFT resistiva de 2.8"

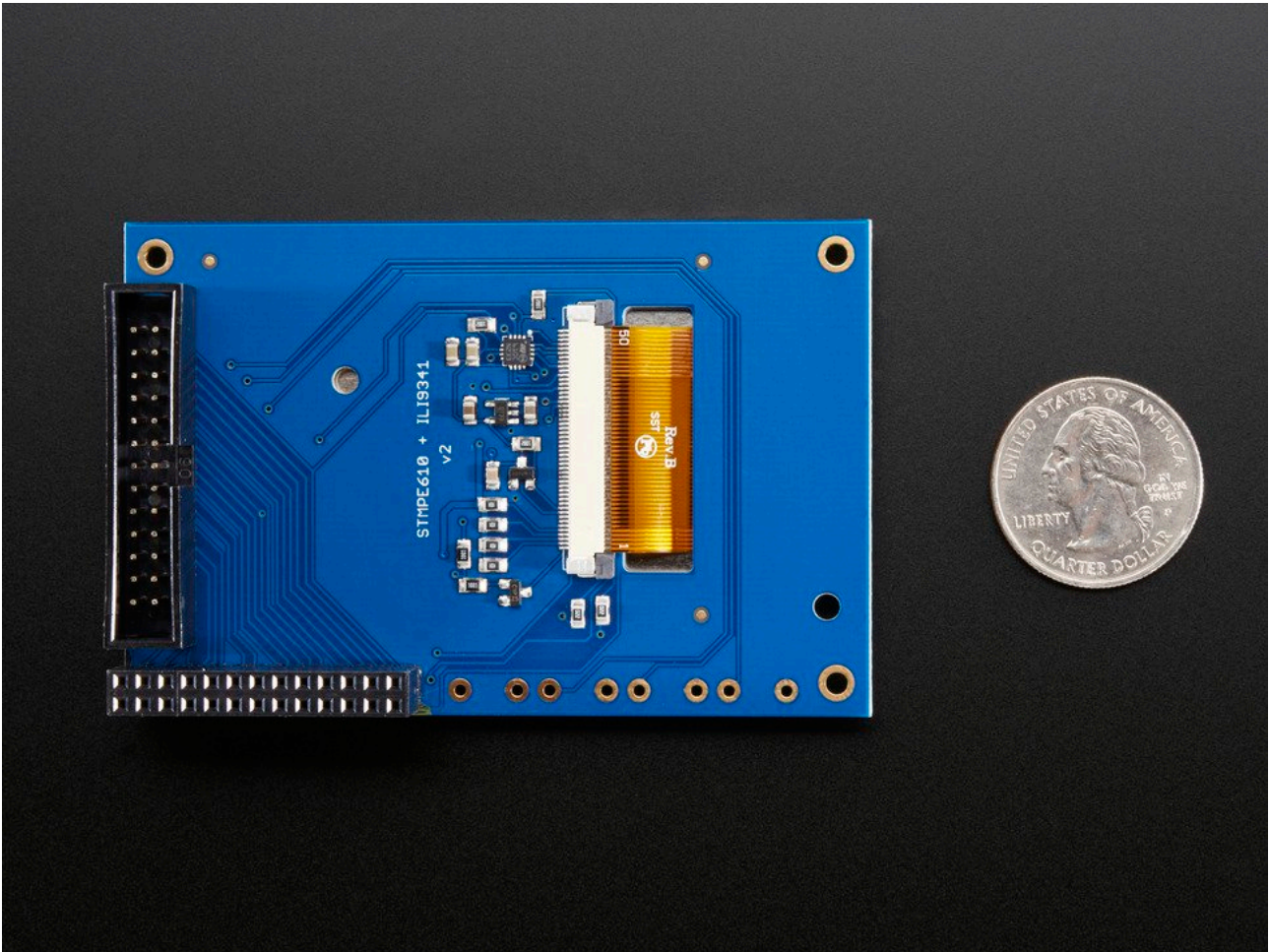


Figura 3.11 Vista frontal posterior de plano de la pantalla PiTFT resistiva de 2.8"

Se trata de un panel táctil ensamblado sobre una pantalla TFT de 2,8 pulgadas. La pantalla viene completamente ensamblada con sus componentes y lista para funcionar con la Raspberry Pi. Se coloca sobre ella, recubriéndola completamente a modo de “sombrero”. Para comunicarse con ella, se emplean los pines correspondientes a SPI (SCK, MOSI, MISO, CE0, CE1) así como los pines #25 y #24 de GPIO. El resto de pines GPIO no se usan. El fabricante dejó espacio para 4 botones táctiles. Estos cuatro espacios están cableados directamente con un pin GPIO cada uno. Esto permite agregar ciertas funcionalidades adicionales (como apagado o reinicio) en caso de desearlo, permitiendo así la construcción de una interfaz de usuario muy básica. En este TFG no se ha usado esta funcionalidad.

El hecho de que use estos pines y no otros es muy importante, pues puede que haya conflicto con el GPS. Esto fue así en un primer momento, pero se resolvió con un conversor USB-serie.

Antes de conectar la pantalla, se procede a iniciar la RPI y descargar desde los servidores del fabricante el script que instala los controladores de pantalla y la configura adecuadamente. Para ello, es necesario introducir en la consola:

<https://forums.adafruit.com/viewtopic.php?f=47&t=122926&p=629573#p629573>

```
$ cd ~
$ wget https://raw.githubusercontent.com/adafruit/Adafruit-PiTFT-Helper/master/adafruit-pitft-helper2.sh
$ chmod +x adafruit-pitft-helper2.sh
$ sudo ./adafruit-pitft-helper2.sh
```

Una vez descargado y ejecutado el script, se accederá a este diálogo:

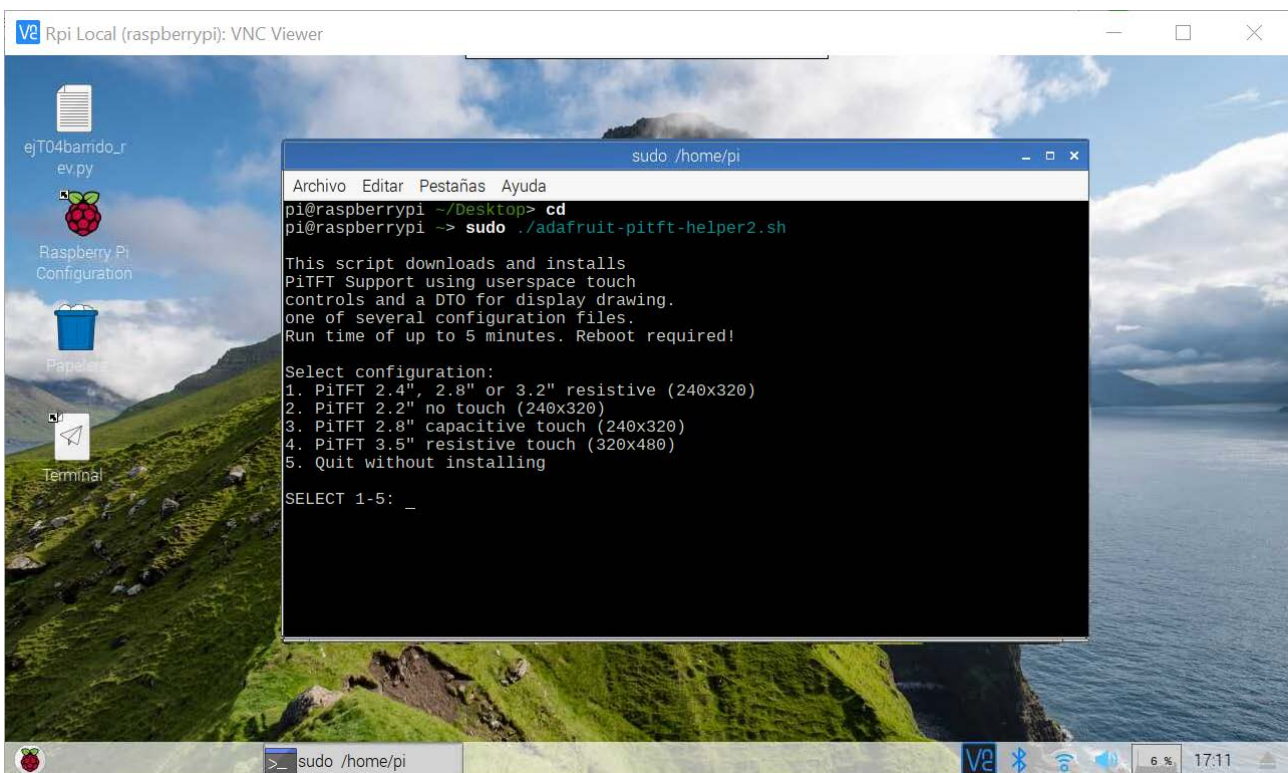


Figura 3.12 Selección de tipo de pantalla

Se tecldea 1 y se confirma con un retorno de carro <Enter>:

Se tecldea 1 ó 3 por si se desea invertir la imagen en la pantalla. En ambos casos es apaisada:

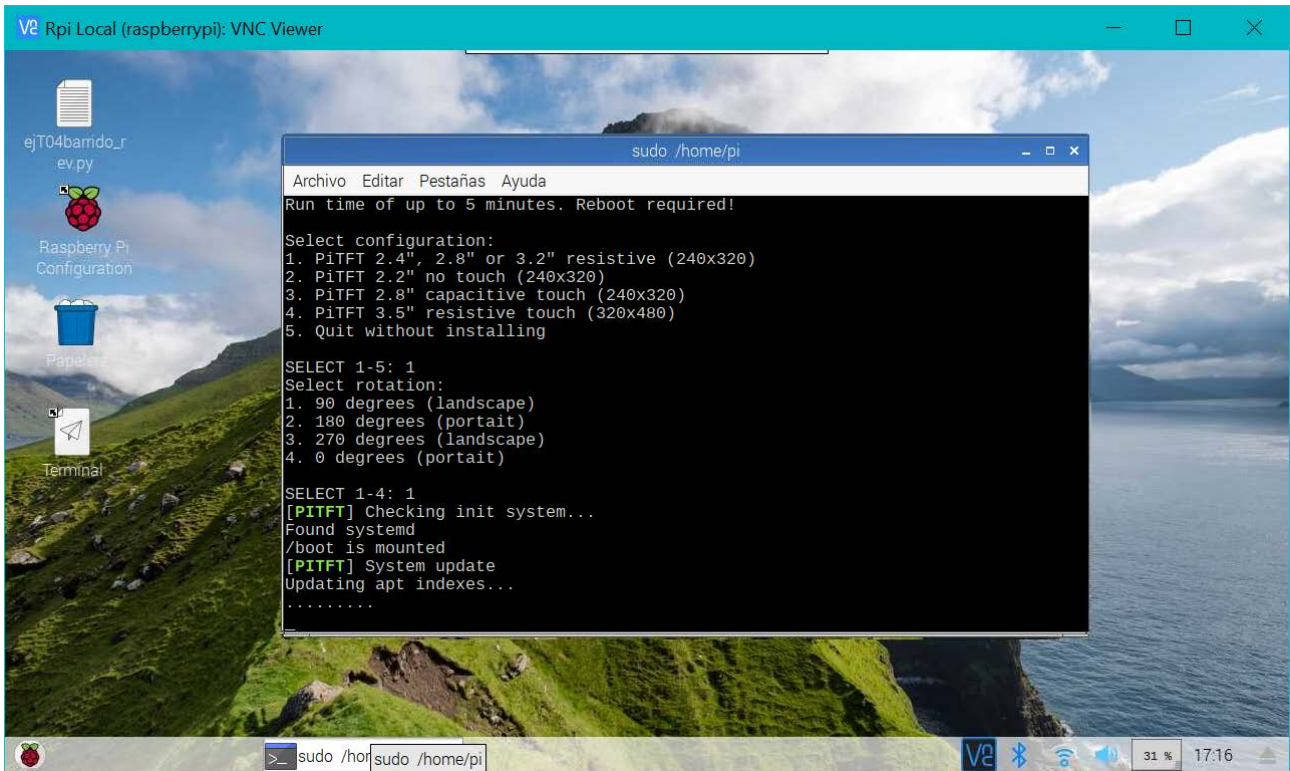


Figura 3.13 Selección formato apaisado no invertido e instalación y configuración del software de la pantalla.

El proceso de configuración de los drivers es completamente automático.

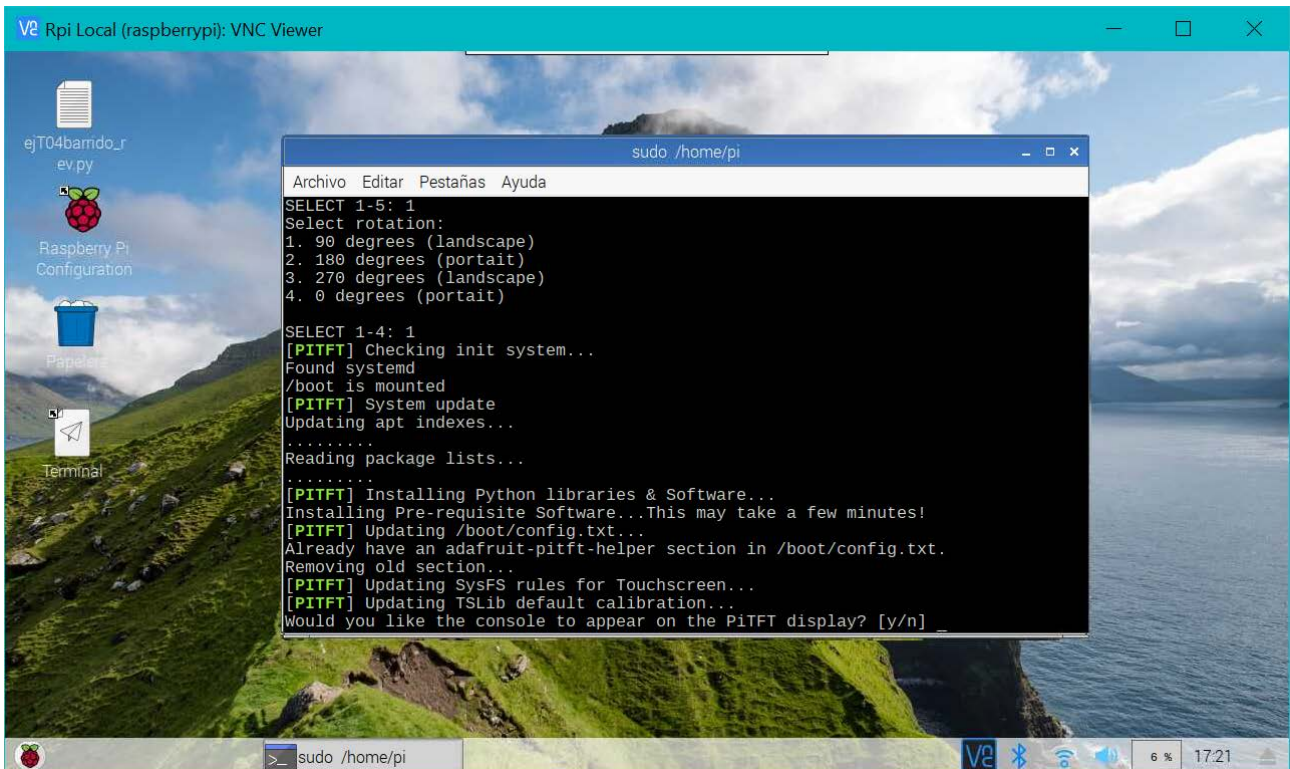


Figura 3.14 Selección modo gráfico/console.

A continuación, se ofrece utilizar la pantalla para visualizar el entorno gráfico nada más iniciarse o, en su defecto, iniciar directamente en modo texto con acceso a la consola bash de raspbian. Para este TFG, seleccionaremos **n** y presionaremos <Enter>. Así, al reiniciar con la consola se accederá directamente al modo gráfico con el escritorio Pixel.

De todas maneras, una vez iniciado el modo texto, se podría pasar al gráfico tecleando simplemente:

```
$ startx
```

Por último, se pide al usuario si desea que, en caso de conectar una pantalla HDMI externa (como un televisor o monitor de ordenador), su contenido se vea también en la pantalla PiTFT. Es **crucial seleccionar y** (yes), ya que de lo contrario el escritorio Pixel **no se visualizará** en la PiTFT, aun sin conectar la RPI a un monitor externo.

Una vez finalizada la configuración, se solicitará permiso para reiniciar. Se selecciona **n** (no) y se pulsa <Enter>.

A continuación se apaga la RPI mediante

```
$ sudo shutdown now
```

Y se coloca la pantalla PiTFT correctamente sobre la RPI.

A continuación, se procede a conectar la alimentación. Es importante respetar los requisitos mínimos, como el voltaje (5V, fundamental) o el amperaje (2,4 A). Si no se respeta este requisito, aparecerá un símbolo de rayo sobre la esquina superior derecha del escritorio Píxel al iniciarse. No implica un riesgo inmediato durante su funcionamiento como puede ser un “*overvoltage*” o “*undervoltage*”, pero no es recomendable trabajar sin corriente suficiente, pues el peligro de corrupción de datos es grande.

3.3 Adafruit Ultimate GPS breakout

El propio fabricante de la pantalla dispone también de accesorios como GPS, motores e incluso cables auxiliares. No es barato, pero es uno de los pocos especializados en la plataforma RPI y con un gran equipo de soporte detrás.

El GPS servirá para obtener coordenadas, velocidad y hora en tiempo real durante la sesión de captura. En la figura 3.15 puede observarse el contenido del paquete que se recibe del fabricante:

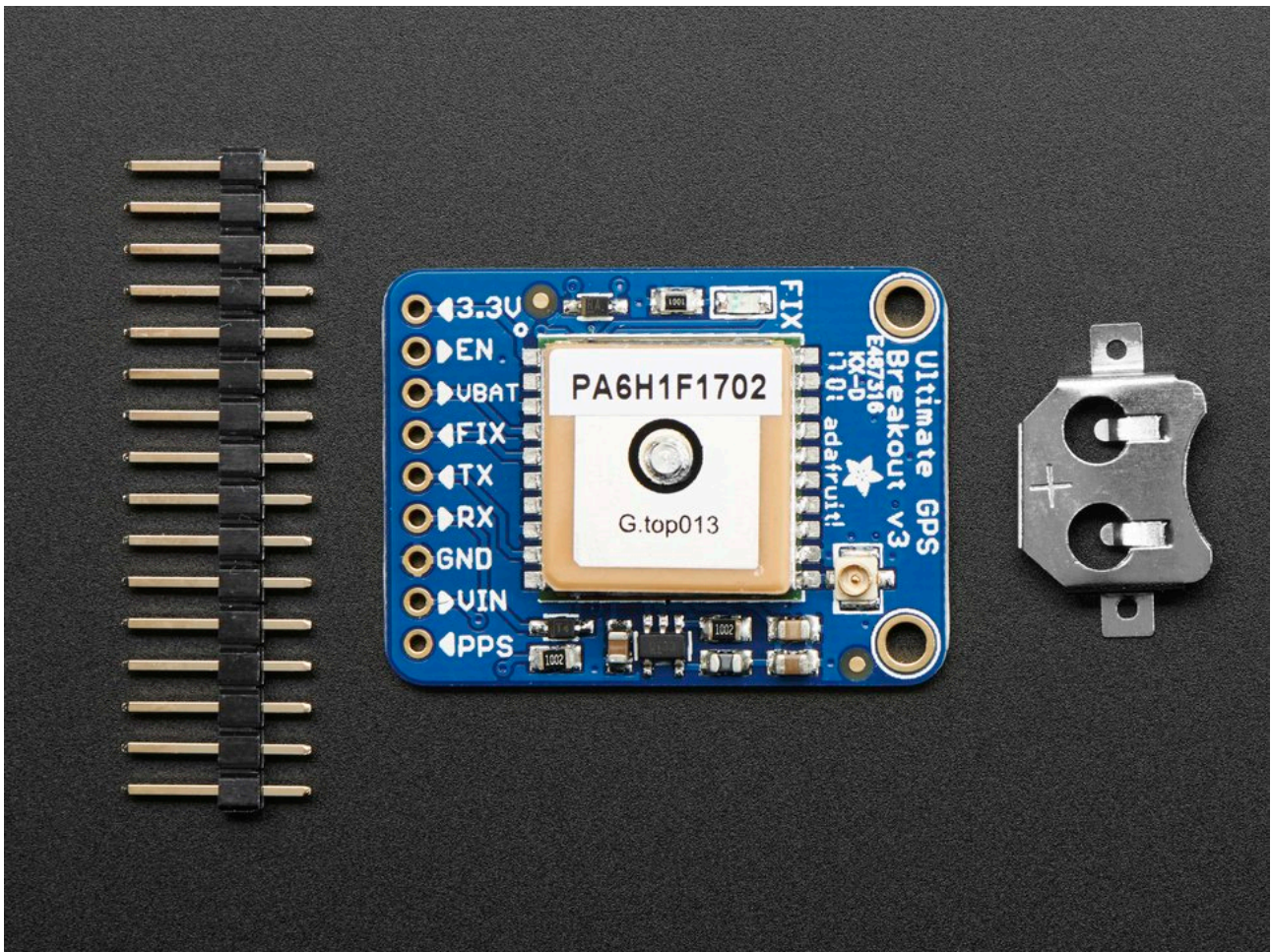


Figura 3.15 Pines para las conexiones (izquierda), módulo GPS y socket para pila de botón (derecha).

El fabricante lo bautizó como Ultimate por ser un todo en uno que satisface las necesidades de la mayoría de usuarios. Entre sus características presenta -165 dBm de sensibilidad, dos orificios de montaje, compatibilidad con batería RTC, puerto para registro de datos (*datalogging*) incorporado y antena interna con conector (salida) para antena externa. El corazón del GPS es el integrado MTK3339, un módulo de GPS de alta calidad que puede rastrear hasta 22 satélites en 66 canales con un receptor de extraordinaria sensibilidad. El consumo de energía es bajo, de tan solo 20 mA. Dispone de alimentación de 3.3-5 VDC y de un pin ENABLE que permite controlar el estado ON/OFF del módulo cuando se usa como periférico de un microcontrolador.

Como detalle, presenta un pequeño LED de color rojo brillante. El LED parpadea con una frecuencia de 1 segundo mientras busca el fix (posición con al menos 3 satélites) y parpadea una vez cada 15 segundos cuando lo consigue para ahorrar energía.

En caso de que se desee disponer de una señal de fix continua, el fabricante proporciona un pin FIX para poder llevar la señal a un led externo.

El módulo GPS en su versión 3 presenta la posibilidad de emplear una antena externa y la capacidad de registrar datos directamente de manera integrada. Estas características no han resultado de relevancia para este TFG.

El módulo viene ensamblado, con un cabezal soldable y soporte para pilas botón CR1220 (para mantener en memoria el valor del RTC). Esta característica tampoco se ha empleado en este GPS.

Conexión. Cable conversor USB-Serie

Puesto que los pines GPIO de la RPI están ocupados, incluyendo el puerto serie, se ha decidido colocar un cable conversor del USB – serie del mismo fabricante, liberando a los pines de esta función y asignándosela a uno de los puertos USB libres.



Figura 3.16 Cable USB – TTL serie de Adafruit.

A continuación se pasa a describir de manera detallada el procedimiento de configuración del cable y la conexión del gps:

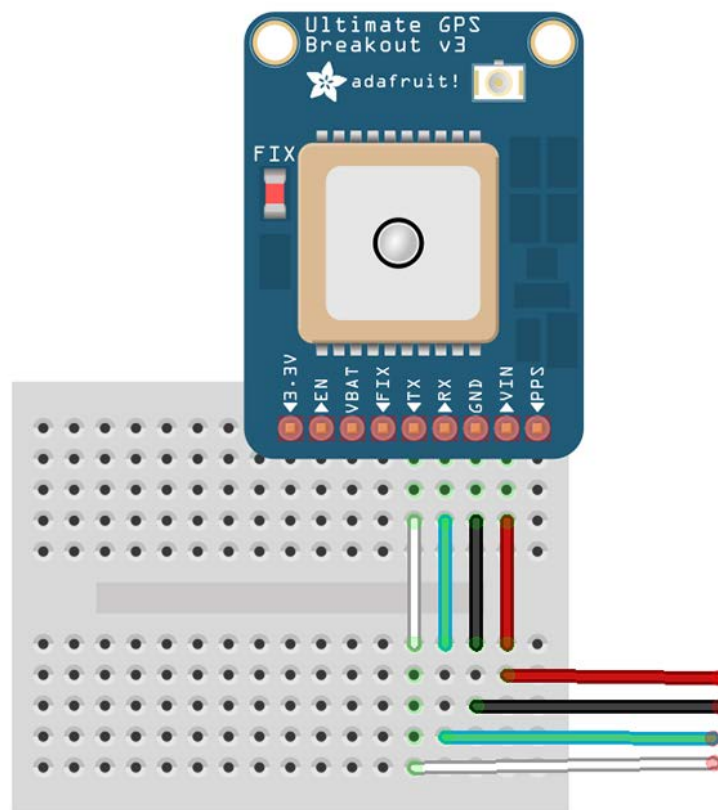


Figura 3.17 Conexión del cable TTL serie – USB con el GPS.

Para comenzar, es necesario conectar los cables tal y como se muestra en la figura 3.17.

Si la conexión es correcta, el LED parpadeará con una frecuencia de 1 segundo. Si la conexión es incorrecta, el LED no parpadeará.

Para comprobarlo, se tecleará en consola:

```
$ ls /dev/ttyUSB*
```

Aparecerá a continuación una lista con todos los dispositivos USB disponibles en la RPI. Lo más normal es que el gps aparezca como /dev/ttyUSB0, si no se tiene conectado cualquier otro dispositivo USB.

Aun así, en caso de tener problemas, basta con introducir:

```
$ sudo lsusb
```

para obtener una lista de todos los dispositivos usb conectados a la RPI, con información de fabricante e identificadores.

Para obtener el contenido “raw” del puerto serie, basta con introducir:

```
$ sudo cat /dev/ttyUSB0
```

Con esto se comprobará que la comunicación es correcta y que funciona adecuadamente. No obstante, para poderlo usar en el software que se ha desarrollado de manera efectiva, es necesario ejecutar un script llamado *startgps.sh*:

```
#!/bin/bash
sudo systemctl stop gpsd.socket;
sudo systemctl disable gpsd.socket;
stty -F /dev/ttyUSB0 raw 9600 cs8 clocal -cstopb;
sudo gpsd /dev/ttyUSB0 -F /var/run/gpsd.sock;
```

Las dos primeras líneas se aseguran de desactivar el servicio gpsd por defecto de Raspbian.

La tercera, de ajustar correctamente la tasa de baudios para que la comunicación sea la adecuada.

La cuarta, se asegura de que el demonio gpsd, ahora ejecutado bajo demanda, se vincule correctamente al GPS conectado a través del cable conversor serie TTL – USB.

Para ejecutarlo, basta con introducir en consola:

```
./startgps.sh
```

Para detener la captura de datos, bastará con ejecutar:

```
./startgps.sh
```

que no contiene más que una línea de código:

```
$ sudo killall gpsd
```

que mata al demonio gpsd que se ejecutó bajo demanda.

Capítulo 4. Herramientas software

En primer lugar, se llevará a cabo una síntesis de las herramientas empleadas. Posteriormente, se expondrán de manera detallada los criterios de selección. Por último, se justificarán varias decisiones de diseño.

4.1 Herramientas software

4.1.1 Lenguajes de programación

Existen dos alternativas fundamentales: Python y C++.

Ambos son lenguajes extremadamente diferentes, y la mayoría de las diferencias no son estrictamente ventajosas en una dirección u otra. Dicho esto, para la mayoría de los usos, resulta fácil proporcionar argumentos a favor o en contra de uno u otro.

Existe también una gran diferencia entre las características proporcionadas por el lenguaje y cómo luego se usan en la práctica. Así, (Murphy, 2012) proporciona 4 categorías donde contrasta las diferencias entre ambos:

Administración de memoria: C ++ no tiene recolección de basura, y favorece el uso de punteros *inteligentes* para administrar y acceder a la memoria. Los punteros desnudos o sencillos se corresponden con los punteros a secas de C (Kieras, 2016) :

```
type * nombre_del_puntero = & nombre_de_variable;
```

Ya desde C ++ 11, se dispone de los llamados "punteros inteligentes". Se llaman "inteligentes" porque ellos mismos eliminan el espacio de memoria ocupado cuando ya no es usado en el programa. Hay 3 tipos de punteros inteligentes en C ++ 11:

```
unique_ptr<typename> nombre_del_puntero;  
weak_ptr<typename> nombre_del_puntero;  
shared_ptr<typename> nombre_del_puntero;
```

En cualquier caso, se trata de una característica relativamente novedosa y por lo tanto no incorporada en código y librerías antiguas.

Además, diferencia entre *heap* (montón) y pila. También distingue entre paso por valor y referencia. C ++ requiere mucha más atención a los detalles de almacenamiento y de iteración sobre elementos. Si bien esto permite un control muy fino, a menudo no necesario (salvo en tareas de ingeniería de sistemas donde resulta imprescindible).

Tipos: los tipos de C ++ se declaran explícitamente, se vinculan a nombres y se comprueban en tiempo de compilación: son **estrictos**. Los tipos de Python están vinculados a valores, se verifican en tiempo de ejecución: son **dinámicos**. Los tipos de Python son también un orden de magnitud más simple. La seguridad, la simplicidad y el hecho de que no haga falta ni declararlos ni comprobarlos en tiempo de compilación favorecen un desarrollo más rápido. No obstante, esto elimina una comprobación adicional que sí se hace en C++; si el usuario no es cuidadoso, pueden enmascarse errores fatales que aparezcan durante tiempo de ejecución del programa. Esto es especialmente peligroso en Python 2, que emplea por defecto codificación ASCII para las cadenas de texto y no Unicode, como sí hace Python 3 por defecto.

Interpretado vs compilado (implementación): C++ casi siempre se compila explícitamente. Python, de manera general, no. Internamente, el código fuente de Python siempre se traduce a *bytecode*, y este es ejecutado por la máquina virtual. Para evitar la sobrecarga que supone analizar y traducir repetidamente módulos que rara vez cambian, el *bytecode* se escribe en un archivo cuyo nombre termina en ".pyc". Esta operación se realiza cada vez que se analiza un módulo. Cuando se cambia el archivo .py correspondiente, se reanaliza, vuelve a traducirse a (Effbot (Python FAQ))y vuelve a escribirse el archivo .pyc. (Effbot (Python FAQ))

Es una práctica común comenzar a desarrollar en el propio intérprete en Python, que es ideal para probar características o fragmentos (*snippets*) de código de manera limpia y rápida. Los desarrolladores de C++ no disponen de esta opción, y el proceso de depuración con herramientas como gdb añade inevitablemente tiempo al desarrollo.

Complejidad del lenguaje: C ++ es muy extenso. Su especificación presenta 775 páginas de jerga legal del lenguaje. Incluso los desarrolladores más experimentados admiten que tan solo usan un subconjunto del lenguaje en su trabajo diario. Python es mucho más simple, lo que permite aprovechar todas sus características. Esto contribuye a un desarrollo más rápido.

Y estas diferencias, a su vez, se derivan de una diferencia general en la filosofía:

C++ trata de proporcionar al desarrollador un gran número de herramientas y, al mismo tiempo, nunca (por la fuerza) abstraer nada que pudiera afectar el rendimiento.

Python intenta emplear una única manera de hacer las cosas y que esta sea lo más sencilla posible, incluso a costa de perder un control más detallado sobre algunos aspectos o eficiencia de funcionamiento. Presenta tipos de datos y estructuras de alto nivel embebidos por defecto en el lenguaje, como diccionarios y listas.

En muchos casos, la filosofía de Python es una ventaja porque permite llevar a cabo la mayoría de tareas de manera más fácil y rápida. También (y esta ha sido la experiencia durante el desarrollo de la aplicación) permite que la consulta de documentación sea más rápida: si solo hay una forma de hacer las cosas, bastará entonces con mirar una sección específica de la ayuda para avanzar rápidamente. Además, muchos otros desarrolladores previamente habrán intentado resolver un problema de forma similar. Esto repercute de manera muy positiva en el rendimiento del programador.

Por supuesto, también comparten muchas similitudes: ambos lenguajes permiten trabajar cómodamente en el paradigma OO; ambos proporcionan soporte para la programación imperativa (con algunas características funcionales, como las expresiones lambda); ambos tienen excepciones (usadas extensivamente a lo largo del código este proyecto, permitiendo un control muy fino de errores) y vastísimas librerías para desarrollar cualquier tipo de software.

4.1.2 Librerías gráficas

En cuanto a qué librería usar para la GUI, se plantearon varias alternativas: Qt (PyQt/PySide), Tkinter y wxPython.

Tkinter (PSF, 2018) es la librería gráfica oficial de la biblioteca estándar de Python. Proporciona un conjunto de herramientas de ventanas robusto e independiente de la plataforma, estando disponible para los programadores de Python que utilizan el paquete tkinter, así como su extensión, los módulos tkinter.tix y tkinter.ttk.

El paquete tkinter es una capa delgada orientada a objetos en la parte superior de Tcl/Tk. Para utilizar tkinter, no se necesita escribir el código Tcl,

pero requiere consultar la documentación de Tk y, ocasionalmente, la documentación de Tcl. Tkinter es, en definitiva, un conjunto de *wrappings* que implementan los *widgets* Tk como clases de Python. Además, el módulo interno `_tkinter` proporciona un mecanismo seguro que permite que Python y Tcl interactúen.

Entre sus virtudes principales, tkinter destaca por ser rápida, fácil y ligera. No obstante, su documentación estándar es débil, y la librería subyacente no emplea el paradigma de orientación a objetos de forma nativa. Por otra parte, sí que hay disponible un buen material *no oficial*, incluyendo tutoriales y hasta un libro. Presenta una apariencia un tanto rústica, y aunque ha mejorado mucho en su versión Tk 8.5, sigue estando un paso por detrás de otras librerías como GTK o Qt.

A continuación se muestra un ejemplo de GUI diseñada con Tkinter:

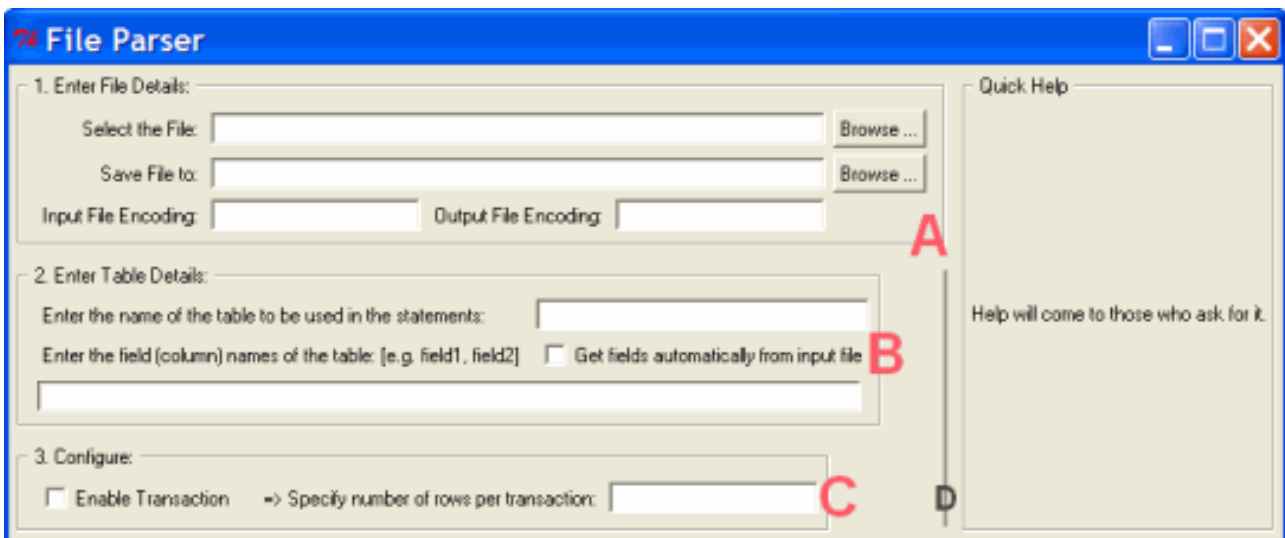


Figura 4.1 Ejemplo de GUI diseñada con Tkinter. Aplicación para leer y codificar archivos. (Sam, 2011)

Qt es un framework de desarrollo de aplicaciones multiplataforma para PC, sistemas empujados (*embedded*) y móviles. Puede trabajar sobre Windows, Linux, OS X, BlackBerry, Android e iOS, entre otros.

Qt está escrita en C++. Un preprocesador, MOC ('Meta-Object Compiler'), se utiliza para extender el lenguaje C++ con funciones como *Signals* y *Slots*. Antes del paso de compilación, MOC analiza los archivos de origen escritos en Qt-extended C++ y genera código fuente C++ estándar compatibles de ellos. Por lo tanto, el propio framework y las aplicaciones/librerías que lo utilizan pueden ser compilados por cualquier compilador compatible con C++ estándar como Clang, GCC, ICC, MinGW y MSVC.

El desarrollo de Qt se inició en 1990 gracias a los programadores noruegos Eirik Chambe-Eng y Haavard Nord. Su compañía, Trolltech, vendió licencias QT y proporcionó soporte, cambiando el nombre de la empresa a lo largo de los años. Hoy Trolltech se llama *The Qt Company* y es una subsidiaria de Digia Plc., Finlandia. Aunque *The Qt Company* es la principal impulsora detrás de Qt, Qt ahora es desarrollado por una alianza más grande conocida como *The Qt Project*, formado por empresas e individuos de todo el mundo.

Qt viene con su propio entorno de desarrollo integrado (IDE), llamado *Qt Creator*. Se ejecuta en Linux, OS X y Windows y ofrece finalización de código inteligente, resaltado de sintaxis, un sistema de ayuda integrado, depurador y también dispone de integración con todos los sistemas de control de versiones más usados (p. ej. git, Bazaar) (Company, 2018). Además de los desarrolladores de Qt Creator en Windows, también puede utilizar el complemento de Visual Studio de Qt. También se pueden utilizar otros IDE (p. ej., KDevelop en KDE).

Con Qt, las GUI se pueden escribir directamente en C++ usando su módulo Widgets. Qt también viene con una herramienta gráfica interactiva llamada *Qt Designer* que funciona como un generador de código para las GUI basadas en Widgets. *Qt Designer* puede usarse de manera independiente, pero también está integrado en Qt Creator. Su uso facilita enormemente la tarea de crear interfaces de usuario coherentes de una manera sistemática. (Company, 2018).

4.1.3 Librería Python-OBDD

Este módulo de Python se concibió como *framework* para posibilitar la extracción de datos de los sensores de un coche mediante el protocolo OBD-II. Es compatible con la mayoría de adaptadores estándar de la industria y dispone de unas tablas con PIDs estándar comunes a la mayoría de vehículos. Permite hacer peticiones usando el módulo *pyserial*. *Pyserial* es un paquete de Python que permite emplear comunicación serie desde Python, proporcionando una API que posibilita al desarrollador abstraerse de los detalles de plataforma. Además, permite devolver no solo los valores numéricos sino también sus unidades, empleando para ello el módulo *Pint* (*Pint* es un paquete de Python diseñado para definir, realizar operaciones y manipular magnitudes físicas como objetos, con su módulo y su unidad asociada). Ha sido diseñada por un entusiasta del software libre, Brendan Whitfield (Brendan, 2016), a partir de un *fork* de *pyobd* (la otra gran

alternativa disponible en la librería estándar de Python). No obstante, el código base se ha revisado por completo, optimizando el uso de la librería estándar de Python y ampliando el número de sensores compatibles. También se ha documentado a fondo.

A continuación, se observa cómo obtener información por OBD de una forma sumamente sencilla mediante el uso de este *framework*:

```
import obd

connection = obd.OBD()

cmd = obd.commands.RPM

response = connection.query(cmd)
print(response) # "2410 RPM"
```

La primera línea se encarga de importar la librería *obd* al espacio de nombres (*namespace*) actual.

La segunda línea se encarga de crear y asignar una conexión *obd* a la variable *connection*.

La tercera línea se encarga de almacenar el comando para obtener el número de revoluciones por minuto (RPM).

La cuarta obtiene la respuesta empleando el método *query* del objeto *obd* asignado en la variable *connection*.

La quinta y última se encarga de mostrar por consola el valor obtenido.

4.1.4 Librería GPSD

Gpsd es el demonio que sirve para obtener datos de GPS, radios, GPS diferenciales o receptores AIS conectados a un host mediante el protocolo USB o RS232 (serie).

El demonio puede configurarse para ejecutarse justo al inicio, o puede controlarse mediante comandos específicos por medio de un socket local. Así, una vez reconocido el dispositivo a monitorizar, GPST ajusta el protocolo adecuado y la velocidad del puerto de comunicación.

Dado un dispositivo GPS por cualquier medio, *gpsd* descubre la velocidad del puerto y el protocolo correctos para ello.

Gpsd es compatible con cualquier GPS que emplee el protocolo NMEA 0183, u otros protocolos NMEA extendidos. En cualquier caso, proporciona soporte para la enormísima mayoría de dispositivos GPS comerciales.

Gpsd busca abstraer las diferencias entre receptores GPS y los diferentes protocolos que usan para proporcionar una interfaz única al desarrollador. También permite el uso de comandos para sintonizar los dispositivos GPS con el objetivo de disminuir la latencia. (S. Raymond).

4.1.5 Protocolo NMEA

La NMEA (Asociación Nacional de Electrónica Marina de Estados Unidos) ha definido una especificación que establece cómo deben comunicarse entre sí las diferentes piezas de los componentes electrónicos usados en la marina. Este estándar permite a estos dispositivos intercambiar información tanto entre ellos como con ordenadores.

La información que proporciona un GPS cae, por lo tanto, dentro de esta categoría. La mayor parte de software que trabaja con obtención de datos en tiempo real espera que estos estén en formato NMEA. Aquí se incluye la PVT (Posición, Velocidad, Tiempo) calculada por el receptor GPS.

La idea básica de NMEA es enviar sentencias de forma sucesiva, cada una independiente de la anterior. Existen sentencias comunes al estándar y otras específicas del fabricante, siendo estas últimas opcionales y variables en función del dispositivo.

De manera general, todas presentan un prefijo de dos letras que sirven fundamentalmente para identificar al dispositivo (GPS o de otro tipo) que usa ese tipo en concreto. A continuación, otra secuencia de tres letras proporciona información sobre el contenido de la sentencia en sí. En caso de que se trate de una específica del fabricante, empezará por P y seguirán tres letras que identifican al fabricante del dispositivo. (DePriest, 2012)

Cada sentencia se inicia con '\$' y termina con un carácter '\n', sumando en total 80 (sin tener en cuenta este último). Los datos dentro de cada sentencia son únicos y están separados por comas. A veces un conjunto de datos puede repartirse entre varias sentencias. Los programas que utilicen estos datos solamente usan la coma para separarlos. La longitud o precisión decimal se determina por medio del punto.

Al final de los campos de datos se incluye un número de comprobación de suma, que puede ser verificado o no por el dispositivo que lee los datos. Este

último campo de *checksum* consta de un '*' y dos dígitos hexadecimales (8 bits en total), obtenidos mediante una operación de OR exclusivo entre todos los caracteres de la sentencia (exceptuando '\$' y '*'). La obligatoriedad o no de su uso dependerá del tipo de sentencia en concreto. (DePriest, 2012)

La versión actual del estándar es la 3.01, pero las que implementan la mayoría de GPS disponibles en el mercado son la 1.5, 2.0 – 2.3. El Ultimate GPS de Adafruit (Fried, 2016) se supone compatible con estas ya que el demonio *gpsd* no ha tenido ningún problema en leer y descifrar la información manejada por el dispositivo GPS.

Conexión Hardware

La interfaz de hardware para unidades GPS está diseñada para cumplir con los requisitos NMEA. También son compatibles con la mayoría de los puertos serie de la computadora que utilizan protocolos RS232, sin embargo, estrictamente hablando, el estándar NMEA no es RS232.

La velocidad de interfaz se puede ajustar en algunos modelos, pero el estándar NMEA es de 4800 bps, con 8 bits de datos, sin paridad y un bit de stop. Todas las unidades que entienden NMEA deberían ser compatibles trabajando a esta velocidad.

A esta velocidad se pueden enviar datos más que suficientes para rellenar un segundo. Es por esto que algunos dispositivos solo envían actualizaciones cada dos segundos o envían algunos datos cada segundo y otros con menos frecuencia. Además, algunas unidades pueden enviar datos con un par de segundos de antigüedad, mientras que otras unidades pueden enviar datos que se recopilan dentro del segundo que se envían.

A 4800 bps solo se pueden enviar 480 caracteres/s. Como una sentencia NMEA puede tener hasta 82 caracteres, el límite teórico máximo de sentencias que pueden enviarse es inferior a 6. El límite real está determinado por las sentencias específicas utilizadas.

En general, están pensadas para recibirse en segundo plano en forma de flujo de datos continuo, siendo consultadas bajo demanda por el software en cuestión. Algunos programas no pueden hacer esto. Como alternativa, tomarán muestras del flujo de datos, luego los usarán para mostrar por pantalla y posteriormente volverán a muestrearlos. Dependiendo del tiempo necesario para usar los datos, puede haber un retraso de 4 segundos en la capacidad de respuesta a los datos modificados. Esto puede ser suficiente para algunas aplicaciones pero totalmente inaceptable en otras.

El estándar NMEA ha existido por muchos años (1983) y ha sufrido varias revisiones. El protocolo ha cambiado y el número y los tipos de sentencias pueden ser diferentes según la revisión. La mayoría de los receptores GPS entienden el estándar conocido como 0183 versión 2. Este estándar establece una tasa de transferencia de 4800 bps. Algunos receptores también son compatibles con los estándares más antiguos, que transferían datos a 1200 bps. Otras unidades de marcas como Garmin se pueden configurar a tasas de 9600 bps o incluso superiores, pero esto solo se recomienda si se ha determinado que a 4800 bps la unidad funciona bien en primer lugar. En el caso del Ultimate GPS Adafruit, recomienda 9600 por defecto. Es importante conocer la tasa máxima de transferencia, puesto que permite configurar el dispositivo para que se ejecute lo más rápido posible, mejorando la capacidad de respuesta del programa.

Para usar la interfaz de hardware se requerirá de un cable. En general, el cable es particular para cada caso, por lo que será necesario disponer de uno fabricado para la marca y el modelo de la unidad que se vayan a usar. Hoy en día los ordenadores ya no incluyen un puerto serie, sino solo un puerto USB, por lo que la mayoría de los receptores gps funcionarán con adaptadores serie a USB y puertos serie conectados mediante un adaptador PCMCIA (tarjeta de PC). Incluso aun disponiendo físicamente de un puerto serie como es el caso de la Raspberry Pi, puede que esté ocupado por otro periférico o simplemente no sea conveniente liberarlo para este uso, por lo que los adaptadores USB-serie están a la orden del día para la mayoría de aplicaciones. Adafruit proporciona uno para comunicarse con el GPS, compacto y con amplia documentación. Para un uso general con un receptor GPS, solo necesitará una línea de RX de datos, otra de TX, alimentación (VIN) y tierra. En caso de querer que el receptor acepte datos en este cable (p.ej. para enviar datos DGPS al receptor), se necesitará una tercera línea para datos, *Data in*.

¿Por qué usar GPSD?

La mayoría de dispositivos GPS disponibles para Raspberry Pi se conectan vía serie. La obtención de datos mediante lectura directa del puerto serie (mediante un comando *cat*, por ejemplo) presenta dos problemas:

a) proporciona acceso a las sentencias NMEA desnudas, sin tratar. Para solucionarlo, GPSD dispone de una interfaz en JSON que muestra los datos de manera legible para un ser humano.

b) es posible que dos aplicaciones quieran obtener simultáneamente información del receptor GPS. GPSD actúa como intermediario para evitar que colisionen o se corrompan los datos.

En la práctica, GPSD proporciona una funcionalidad adecuada pero muy austera en cuanto a características de alto nivel, por lo que se suelen emplear librerías como GPS3 que proporcionan una API agradable para poder obtener datos como velocidad, latitud, longitud o altura de manera cómoda y sencilla, encargándose la propia librería de gestionar las peticiones y el intercambio de información con el demonio GPSD.

4.1.6 Librería GPS3

Se trata de una interfaz de Python 2.7 – 3.5 para el módulo GPSD. Consta fundamentalmente de dos clases: GPSSocket y DataStream.

```
1. from gps3 import gps3
2. gpsd_socket = gps3.GPSSocket()
3. data_stream = gps3.DataStream()
4. gpsd_socket.connect()
5. gpsd_socket.watch()
6. for new_data in gpsd_socket:
7.     if new_data:
8.         data_stream.unpack(new_data)
9.         print('Altitude = ', data_stream.TPV['alt'])
10.        print('Latitude = ', data_stream.TPV['lat'])
```

GPSSocket crea una conexión socket GPSD y realiza peticiones para obtener información del propio demonio GPSD (línea 2). Se conecta y permanece a la espera (líneas 4 y 5).

DataStream se encarga de seleccionar la información de todo el flujo de datos (altitud, latitud...) que circula por el socket y almacenarla, bien como diccionarios de python (gps3), bien como atributos de objeto (agps3).

```
from gps3 import agps3
gpsd_socket = agps3.GPSSocket()
data_stream = agps3.DataStream()
gpsd_socket.connect()
gpsd_socket.watch()
for new_data in gpsd_socket:
    if new_data:
        data_stream.unpack(new_data)
        print('Altitude = ', data_stream.alt)
        print('Latitude = ', data_stream.lat)
```

El uso de una u otra es análogo y no conlleva más que una preferencia personal por el formato de los datos, pues ambos beben de la misma fuente (los datos en formato JSON proporcionados por el demonio `gpsd`), presentándose de una manera u otra en función de la clase usada.

4.1.7 Simulador *obdsim*

Puesto que disponer de la presencia física de un coche para este caso ha resultado harto complicado, **obdsim** es un simulador de PID y DTC que proporciona un método sencillo para comprobar el funcionamiento del software desarrollado.

Se lleva a cabo mediante la siguiente sentencia:

```
$ obdsim -g gui_fltk &
```

Esto lanzará una interfaz similar a la mostrada en la figura 4.2, donde será posible establecer unos valores de RPM, Posición del pedal acelerador, Temperatura del motor y flujo masa aire (MAF), así como velocidad bajo demanda del usuario, para comprobar si el programa se conecta correctamente.

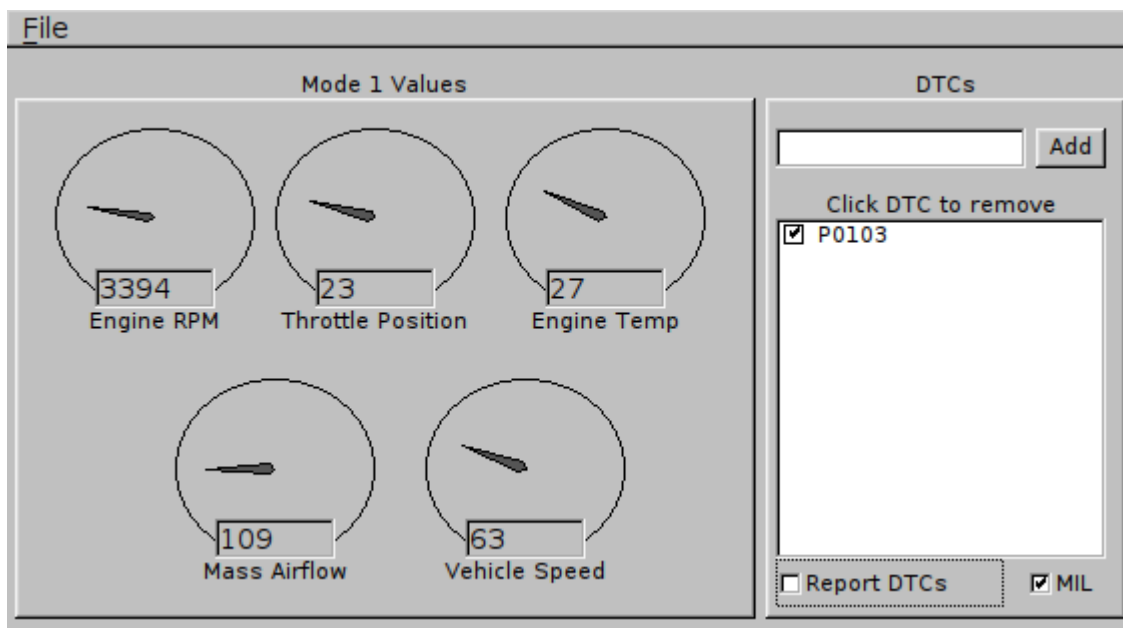


Figura 4.2 Interfaz gráfica del simulador *obdsim*

4.1.8 Simulador *gpsfake*

En línea con lo expuesto anteriormente en el punto 4.1.7, puede darse un caso análogo con la señal de *fix* (conexión satelital) del GPS. En interiores, pese a la gran calidad de la antena incorporada, resulta muy difícil obtener triangulación de la señal

o datos del almanaque. Con este fin, junto con `gpsd` se suele distribuir una pequeña aplicación llamada `gpsfake`, que se encarga de leer un archivo de log NMEA de una sesión real de GPS previa (o generada automáticamente) y mostrarla como si de una captura de datos en tiempo real se tratara. Es tremendamente útil para probar el funcionamiento de código que incluya características de geolocalización u obtención de datos por GPS.

Para instalarlo basta con ejecutar:

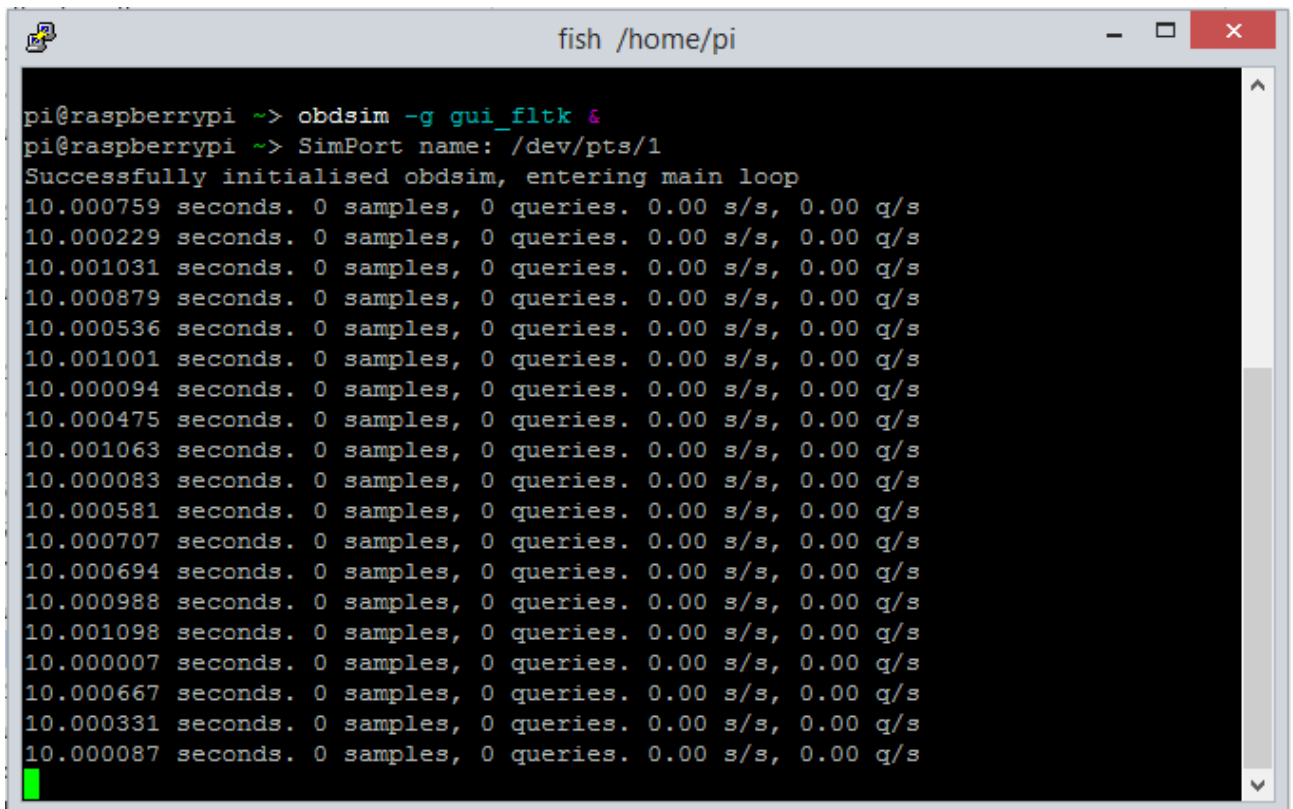
```
$ sudo apt-get install gpsfake
```

Y para llamarlo, sencillamente:

```
$ sudo gpsfake -s 9600 -c 1 stockholm_walk.nmea &
```

Este simulador lleva en sus parámetros un ajuste de la tasa de baudios a 9600 (s se refiere a *speed*) y c a 1 (que el ciclo se repita cuando el simulador termine de reproducir la información procedente de la última línea del log que está leyendo). El carácter & envía la ejecución del proceso a segundo plano.

El archivo `stockholm_walk.nmea` no es más que un conjunto de sentencias NMEA separadas por retornos de carro y mostradas periódicamente por la aplicación `gpsfake`.



```
fish /home/pi
pi@raspberrypi ~-> obdsim -g gui_fltk &
pi@raspberrypi ~-> SimPort name: /dev/pts/1
Successfully initialised obdsim, entering main loop
10.000759 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000229 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.001031 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000879 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000536 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.001001 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000094 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000475 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.001063 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000083 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000581 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000707 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000694 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000988 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.001098 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000007 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000667 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000331 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
10.000087 seconds. 0 samples, 0 queries. 0.00 s/s, 0.00 q/s
```

Figura 4.3 Salida de consola del simulador `obdsim`

Cabe destacar, que en vez de un puerto bluetooth se emplea un puerto serie virtual localizado en /dev/pts/x, donde x es un número natural.

El software también puede ejecutarse en modo Cycle, donde los parámetros varían de manera cíclica entre sus valores mínimos y máximos, de paso en paso.

```
$ obdsim -g Cycle &
```

Así, si se ejecuta:

Los parámetros se moverán entre sus valores mínimos y máximos. Esto incluye TODOS los pids soportados por el modo anterior, y no sólo los pocos del modo gráfico.

Por otra parte, el número de puerto /dev/pts/x se usará como argumento en la inicialización del software.

Así, para ejecutar el software del TFG, basta con realizar:

```
$ cd
$ cd ./TFGv3
$ python __main__.pyw /dev/pts/x
```

Donde x es, como siempre, el número de puerto virtual que se está emulando.

4.1.9 Librería SQLite

SQLite es una librería de software que proporciona un sistema de gestión de base de datos relacional. El consumo de recursos del sistema es muy contenido, de ahí su nombre (*Lite*). Presenta las siguientes características: sin necesidad de usar servidor, auto-contenida, sin necesidad de configuración inicial y transaccional (SQLite Tutorial, 2016).

- **Sin necesidad de usar servidor:** normalmente otros sistemas de BBDD relacionales requieren un proceso servidor que se ejecute por separado para funcionar. Es decir, operan bajo una arquitectura de cliente-servidor. Es el caso de MySQL, PostgreSQL,... Así, las aplicaciones que deseen acceder al servidor que contiene la base de datos usarán el protocolo TCP/IP para enviar y recibir peticiones. En la figura 4.4 se muestra un ejemplo de arquitectura cliente - servidor.

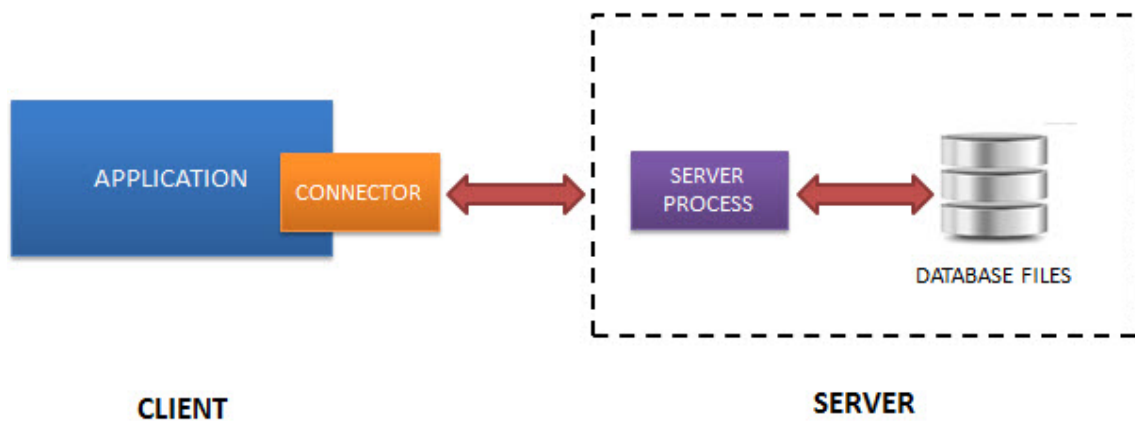


Figura 4.4 Arquitectura cliente - servidor de una base de datos relacional (RDBMS) (SQLite Tutorial, 2016).

SQLite no funciona así. No requiere un servidor para funcionar. De esta forma, la base de datos se lee y se escribe directamente desde la aplicación, tal y como puede observarse en la figura 4.5:

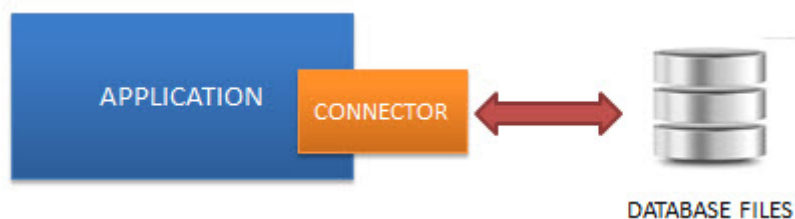


Figura 4.5 Arquitectura de una base de datos SQLite (SQLite Tutorial, 2016).

- **Autocontenida:** SQLite apenas tiene dependencias con sistema operativo o con librerías externas. Esto hace que pueda usarse en multitud de entornos, especialmente sistemas empotrados como videoconsolas, teléfonos inteligentes o, como es el caso, una Raspberry Pi. Está desarrollada en C ANSI, por lo que basta con incluir `sqlite3.c` y `sqlite3.h` al código para poder usar sus características. Está además integrada en la librería estándar de Python, por lo que un sencillo `import sqlite3` al principio del código basta para poder usar sus funciones.
- **Sin necesidad de configuración inicial:** No requiere ni instalación ni archivos de configuración.
- **Transaccional:** SQLite es compatible con ACID. Todos los cambios que se realicen en la base de datos cumplen con los principios de Atomicidad, Consistencia, Aislamiento y Durabilidad; es decir, los cambios que se realizan en una base de datos mediante SQLite o bien se llevan a cabo de manera completa o no se producen, incluso en

entornos donde surjan situaciones inesperadas como una pérdida inesperada de la alimentación.

En definitiva, SQLite proporciona, con una curva de aprendizaje mínima, capacidades de gestión de información avanzadas tan solo importando una pequeña librería en el código que se esté usando. Para este TFG un archivo de datos separados por comas (csv) hubiera bastado, pero el uso de SQLite proporciona al sistema una escalabilidad que facilitará en un futuro continuar con el desarrollo.

4.2 Librería Qt

4.2.1 Modelo de objetos

Qt presenta una clase básica denominada `QObject`. Todas las clases adicionales que se diseñen han de derivarse de ella. De esta forma, las clases hijas heredan las funcionalidades de la librería Qt, como por ejemplo el manejo sencillo de memoria o el sistema de ranuras y señales. A la vez, cabe recordar que Qt es C++ con macros, luego cualquier aplicación Qt admite código en C++ o C.

A continuación, se describen algunas de las propiedades del modelo de objetos Qt.

4.2.2 Manejo sencillo de memoria

Cualquier clase Qt deriva a su vez de una clase anterior, hasta llegar a `QObject`. Cualquier clase hija derivada incluye en su constructor un puntero al objeto padre. Así, si se elimina el objeto padre, también se hará lo propio con los hijos que hayan sido creados. En cambio, en C++ resulta imprescindible destruir cada objeto de forma individual mediante el uso del comando *delete*.

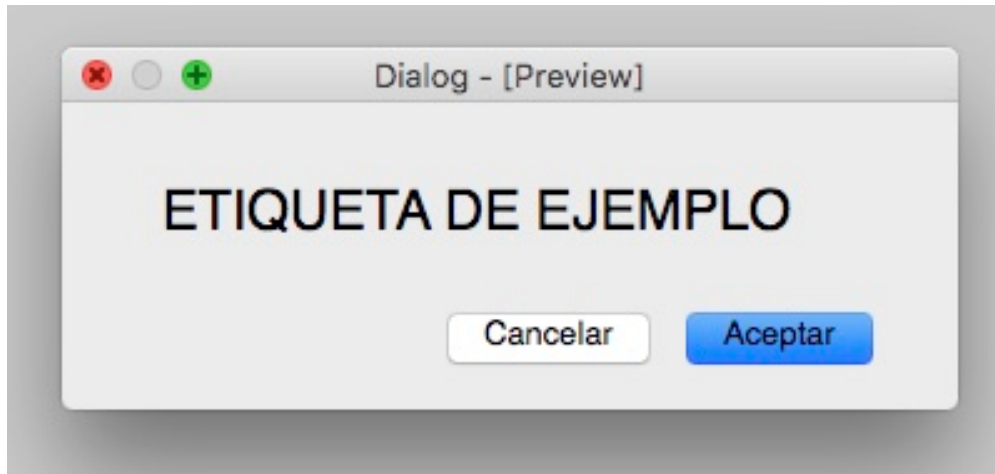


Figura 4.2 Ejemplo de objeto “QDialog”.

La figura 4.2 muestra un ejemplo de aplicación, que consta de un objeto **QDialog** ventana (que constituye el padre) con dos instancias de la clase **QPushButton** y una instancia de la clase **QLabel** (hijos del objeto QDialog). La jerarquía de memoria se representa con más detalle en la figura 4.3.

QDialog es una clase derivada de **QObject** que funciona como molde para gráficos como botones, ventanas y otros ítems.

De la misma forma, **QDialog** es el padre o *parent* de **QPushButton** y **QLabel**, que constituyen sus clases derivadas o hijas.

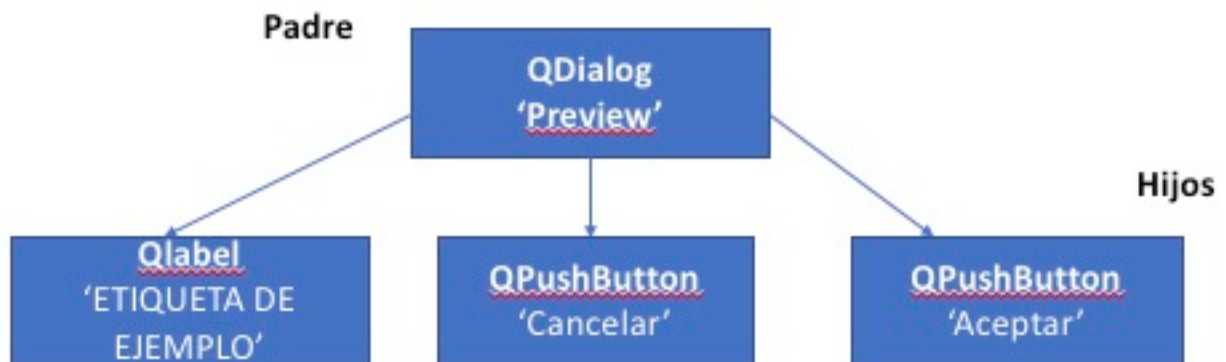


Figura 4.3 Jerarquía de memoria de la aplicación de la figura 4.2.

4.2.3 Signals y Slots

El mecanismo de *signals* (señales) y *slots* (ranuras) permite comunicar los distintos objetos Qt de manera sencilla y lógica para el programador, abstrayendo los detalles de todo el proceso.

- Una señal es equivalente a un *flag*, que se activa al ocurrir un evento, bien predefinido por Qt, como *clicked()* -botón pulsado- o bien especificado por el usuario.
- Un *slot* es la función que se lleva a cabo en el otro objeto, que de nuevo puede estar predefinida por Qt (p. ej. cerrar la ventana) o ser definida por el desarrollador. Esto es lógico: un *slot* no es más que un método miembro de una clase Qt. Si no realiza una de las acciones por defecto predefinidas por la librería Qt, deberá ser el propio usuario quien lo cree.

El mecanismo de señales y ranuras otorga una gran libertad al permitir comunicar dos objetos Qt entre sí, incluyendo el paso de parámetros. La comunicación se lleva a cabo a través del método *connect* que hereda cada ventana Qt por defecto (diálogo, *widget*, aplicación...). Por ejemplo, el usuario puede pulsar un botón de tipo *QPushButton*, emitiendo una señal por defecto *clicked()* y en respuesta, el *slot* procede a rellenar el campo de texto de una etiqueta *QLabel*.

El principal punto a favor de este sistema es que abstrae de los detalles al objeto de Qt que envía la señal: basta con que sepa la interfaz (slot) que lleva asociada. Esto se conoce con el nombre de *loose coupling*, y disminuye enormemente la posibilidad de error. Así, las señales y ranuras desde el punto de vista de la sintaxis del lenguaje no son más que palabras clave que sustituye el *preprocesador* de C++, transformándolas en código, sin necesidad de que el usuario conozca los detalles de todo el proceso. Favorece además un estilo de programación limpio y claro, permitiendo al programador identificar el origen y destino de las señales en la jerarquía de objetos. Por último, facilita enormemente la tarea de depuración, al explicitar claramente la relación entre cada objeto, la señal que envía y la acción que lleva asociada.

4.3 Criterios de selección

4.3.1 Lenguaje de programación

En la decisión final de emplear **Python** han pesado varios factores, en concreto:

- 1.- Su ubicuidad en la plataforma Raspberry Pi asegura una gran cantidad de documentación, ejemplos y comunidad de usuarios.

2.- Gran cantidad de librerías gráficas disponibles para realizar la interfaz de usuario.

3.- Su sintaxis clara, similar al lenguaje natural, con tabulaciones en vez de llaves, asegura una curva de aprendizaje suave.

4.- La filosofía de “pilas incluidas” y la facilidad de extensión de funcionalidad mediante la instalación de paquetes precompilados permite al programador concentrarse en qué se va hacer y no en el cómo.

5.- La existencia de al menos dos librerías con funcionalidad obd en la biblioteca estándar de Python, con multitud de ejemplos y documentación disponible.

6.- La existencia de *bindings* para la mayoría de librerías de C++ y de otros lenguajes, permitiendo el acceso a la funcionalidad completa.

4.3.2 Librería para desarrollo de GUI

En lo que se refiere a la librería para desarrollar la interfaz de usuario, Tkinter presenta una serie de ventajas, como por ejemplo el estar incluida por defecto en la librería estándar de Python. Esto implica que la construcción de ejecutables es mucho menos laboriosa, repercutiendo de manera positiva en el peso de los archivos de código que se distribuyen. Su facilidad de aprendizaje y para ser productivo en poco tiempo contribuyen a considerarla como una opción a tener en cuenta.

No obstante, las limitaciones son dos, pero importantes: no dispone de *widgets* avanzados. Tkinter no viene con *widgets* avanzados que permitan introducir datos de manera compleja (p. ej. Selector de fecha) o que permita una personalización a nivel incluso de píxel como es el caso de Qt. Para aplicaciones profesionales o de nicho, simplemente no da la talla y se necesitan más herramientas: conforme una aplicación crece en tamaño se necesita un constructor de UI (interfaz de usuario) para automatizar la escritura de código, algo imprescindible en entornos de producción. (Slant, 2016)

Entre las ventajas de Qt (PyQt), destacan:

1.- Qt es mucho más que un conjunto de clases C++ adaptadas para construir GUIs: proporciona *wrappings* para librerías nativas de cada plataforma: de esta manera, la interfaz de usuario se adaptará al sistema en el que se ejecute la aplicación.

2.- El mecanismo de *signal/slot* proporciona una flexibilidad casi absoluta: al separar interfaz (evento ocurrido) de implementación (acción a realizar) permite la comunicación de los distintos objetos de la aplicación entre sí de manera muy eficaz mediante el intercambio de mensajes de alto nivel. El código resultante es lógico, limpio, legible y mucho más fácil de mantener.

3.- Estabilidad: Qt se ha usado de manera parcial o total para construir cientos de aplicaciones comerciales muy conocidas. PyQt también ha probado su estabilidad a lo largo del tiempo.

4.- La disponibilidad de un software para construir interfaces de manera gráfica, *Qt Designer*, de tipo WYSIWYG con soporte *drag-and-drop* (Slant, 2016). PyQt dispone de scripts para convertir los archivos **.ui** generados por el Designer en ficheros **.py** ejecutables.

5.- La disponibilidad por defecto de Widgets nativos, incluyendo botones, cajas de texto, menús... con un aspecto nativo adecuado a cada plataforma. Esto garantiza la compatibilidad para pantallas más pequeñas y con el sistema Linux de la Raspberry Pi.

6.- La enorme cantidad de recursos para aprender a usarla, puesto que PyQt es, sin lugar a dudas, una de las librerías más populares para el desarrollo de aplicaciones gráficas en Python.

Por todos estos motivos PyQt es la librería indicada para el desarrollo de la aplicación. No obstante, se listan a continuación (sin ánimo exhaustivo) las desventajas de emplear PyQt:

- Curva de aprendizaje grande. Puede tomar un cierto tiempo aprender a sacar partido a todas las funciones de PyQt.
- PyQt5 es una versión con poca documentación específica aún, lo cual puede confundir a alguien sin experiencia en C++ y Qt. Sin embargo, esto se palia fácilmente empleando la versión anterior (PyQt4) y consultando la infinidad de ejemplos disponibles en Internet.
- Licencia. PyQt está desarrollada por Riverbank, que liberó las librerías bajo doble licencia GPL v3 y una comercial propia de la compañía. Si la aplicación que se va a desarrollar no es de código abierto, es necesario pagar a la compañía para poder vender tu software. Esto no es de relevancia en el ámbito de este TFG, pero se soluciona fácilmente recurriendo a la versión alternativa de PyQt, PySide, que permite distribución del producto software final con ánimo de lucro.

Capítulo 5. Estructura del software

5.1 Descripción

El desarrollo es, lógicamente, la parte que más tiempo ha requerido. Debido al tamaño tan pequeño de la pantalla que se va a utilizar (2'8"), se ha decidido crear distintas pantallas a partir de la clase QDialog de Qt para mostrar toda la información relevante, posibilitando a la vez una interacción adecuada y adaptada a un entorno táctil, con textos y botones grandes.

De esta forma, el software consta de siete diálogos y tres archivos de configuración escritos en formato de texto plano y fácilmente editables, incluso para personas poco familiarizadas con la terminal *bash* de Linux.

En primer lugar, aparece el **diálogo de bienvenida**, donde se presenta un saludo. Acto seguido, se pide seleccionar una opción de las dos disponibles: mostrar una lista de usuarios que hay disponibles en la aplicación o salir. Se trata de un diálogo austero pero eficaz que busca llamar la atención sobre el inicio del programa y presentarle al usuario la interfaz, tipos de letra, botones y estilo con los que va a trabajar a lo largo de una sesión de captura de datos.

Durante el inicio de este diálogo, el software *parseará* de los archivos de configuración *user_list.cfg*, *car_list.cfg* y *pid_list.cfg*. Los dos primeros contienen información relativa al usuario y al vehículo seleccionados, respectivamente. El último archivo contiene la lista de los PID disponibles y que el usuario puede configurar para que sean grabados en una base de datos antes de la sesión de captura. Nótese que, para el registro de estos datos, la selección de PID ha de hacerse previamente en el propio archivo de configuración y no en tiempo de ejecución.

Además, en cuanto se seleccione la opción de *“mostrar lista de usuarios”*, el software realizará automáticamente una prueba de conexión GPS, que será percibida por el usuario mediante un sombreamiento del botón pulsado y un mensaje de espera. Posteriormente, se iniciará el siguiente diálogo con toda la información leída de los ficheros en la ejecución del diálogo de bienvenida.

En segundo lugar, encontramos la pantalla con el **diálogo de inicio de sesión**. En esta pantalla encontramos dos cuadros combinados y un botón de aceptar. La caja superior se ha configurado para que, pulsando y manteniendo sobre cualquier punto de ella, se despliegue un menú en el que aparezca una lista con los usuarios disponibles. A continuación, basta con arrastrar y soltar justo encima del nombre con el que uno se quiera identificar en el software. Se procederá de manera análoga con el cuadro para los vehículos, donde se muestra una lista con todos los vehículos disponibles. Tras realizar la selección, se presionará una única vez el botón de "Aceptar".

Por último, se pasará a realizar una prueba de conexión OBD. Dicha prueba será percibida por el usuario mediante un sombreado del botón pulsado y un mensaje de espera:

- Si la conexión es exitosa, se mostrará un mensaje en el que se indicará que el coche está conectado (es decir, su estado) y la clase de protocolo OBD utilizado. En cuanto sea aceptado, se pasará a la siguiente pantalla, junto con la información relativa al usuario y vehículo seleccionados.
- En caso de error de conexión OBD, se mostrará el estado de conexión del coche y un mensaje de error. En cuanto este sea aceptado, la aplicación terminará su ejecución.

A continuación, se abrirá el **menú principal**. Este menú presenta cuatro botones con una acción distinta cada uno. El primer botón permite iniciar una nueva sesión de captura con la configuración actual; el segundo botón, reconfigurar la sesión de captura, es decir, seleccionar nuevo usuario y vehículo volviendo a la pantalla anterior; el tercer botón, permite escanear los PID del coche actual, o lo que es lo mismo, obtener una lista de los PID que el vehículo seleccionado admite. La lista está limitada a aquellos PID que pueda reconocer la base de datos de la librería OBD de Python. Este paso deberá realizarse única y obligatoriamente la primera vez que se añada nuevo vehículo al archivo de configuración *pid_list.cfg*. Tras ejecutar esta función, se deberá abandonar la aplicación y comentar (#) en la lista de *pid_list.cfg* aquellos que no se desee registrar en la base de datos.

Durante las sucesivas sesiones, no será necesario volver a ejecutar esta función. Por último y en cuarto lugar, el botón "Salir" permite al usuario abandonar la ejecución.

Una vez presionado el botón de nueva sesión, se abrirá el **diálogo de selección** de PID. En esta pantalla se ofrece la posibilidad de seleccionar el PID

que deseamos que sea mostrado en tiempo real durante la sesión de captura. Para ello se dispone de dos botones laterales que permiten desplazarse por la lista de PID leídos del archivo *pid_list.cfg*. Se muestra tanto el nombre del PID según la librería OBD de Python como el valor numérico. Para volver al menú principal basta con presionar el botón de "Atrás"; para continuar, el botón "Aceptar". Una vez presionado, se le pide al usuario que aguarde al inicio de la sesión sombreando el botón de aceptar y mostrando un mensaje de "Espere...".

Por último, aparece el diálogo de **muestra de PID**, formado por dos rótulos y dos botones. Los rótulos describen el nombre del PID que se va a mostrar y su valor en cada instante, respectivamente. Además, y de manera paralela, se consulta el valor de los PIDS leídos y se almacenan cada segundo en una base de datos, única por sesión. Estos valores no son visibles durante la ejecución del software.

En cuanto a los botones, el botón de "Atrás" permite retroceder al diálogo de selección de PID; el botón de "Online" permite abrir la ventana de procesamiento online. Esta última se ejecuta de manera paralela al diálogo de mostrar el PID y el botón lo único que hace es mostrarla u ocultarla según convenga.

La **ventana de procesamiento online** muestra una serie de datos y de características que son interesantes para análisis de la eficiencia en la conducción. Esta serie de características permiten al conductor conocer sus hábitos y saber así cómo modificarlos con el objetivo de buscar una conducción más eficiente. Los criterios se han obtenido conforme a la información proporcionada por el tutor del TFG y buscan ser una herramienta que permita al usuario del software ahorrar combustible y mejorar la seguridad en la conducción.

El primer criterio es el valor de rpm óptimo. Este valor variará en función de si es un coche diésel, gasolina o eléctrico. Para coches de gasolina el rango está entre 1300 y 2000 rpm; para coches diésel el rango está entre 1500 y 2500 rpm. Si el coche no permanece dentro de estos intervalos, se aumentará en una unidad por segundo el contador asociado a la derecha hasta que se corrija.

El segundo criterio es el *throttle level* o nivel de pisado del acelerador. Se busca que este no supere 70%. Si supera este valor, el contador de la derecha asociado aumentará en una unidad una vez por segundo hasta que se corrija.

El tercer criterio es el tiempo que tarda el coche en llegar de 0 a 50 kilómetros por hora. Si este tiempo supera los 5 segundos, el contador asociado de la derecha aumentará en una unidad.

El cuarto criterio es el tiempo de parada, es decir, el tiempo que tarda el coche en detenerse por completo desde el último máximo de velocidad. Es decir, se mide el tiempo que tarda en llegar a cero la velocidad desde el último valor máximo. El valor que se muestra a la derecha representa el último tiempo de parada.

El quinto y último criterio es la desviación típica de la velocidad en los últimos 60 segundos, es decir, cuánto ha variado la velocidad durante ese tiempo.

A continuación, se pasará describir con mayor detalle el código de cada apartado, incluyendo diagramas UML que explican su funcionamiento.

5.2 Diálogo de bienvenida

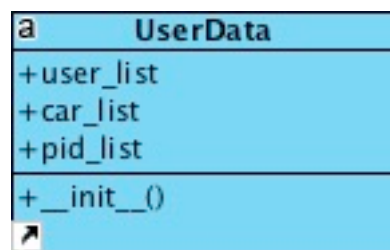


Figura 5.1 Diagrama de clase de 'UserData'.

El archivo *WelcomeDialog.py* contiene dos clases de usuario importantes: *UserData* (tal y como puede verse en la figura 5.1) y *WelcomeDialog*.

La primera clase es una especie de envoltorio para almacenar los datos de la lista de usuarios, la lista de coches y lista de PID que se usarán para diferenciar y almacenar la información recopilada de la sesión. Esto se lleva a cabo al instanciarse en su método `__init__()`, empleando para ello la librería *ConfigParser* de Python. Esta librería se utiliza desde el módulo *cfgparser.py*, presente en la carpeta *data* y diseñado para ese propósito. Este módulo incluye varias funciones que, estrictamente hablando, no son necesarias para el correcto funcionamiento del proceso de *parseo* de datos (p.ej. `add_user()`, `remove_user()`, `add_car()`, `remove_car()` o `add_pids()`). Esto se hace con ánimo de automatizar el proceso de añadir grandes cantidades de información a los archivos de configuración, con un formato correcto; también, con ánimo de completitud, es decir, buscando completar la funcionalidad de *parseo*

desde el punto de vista de la interfaz. De esta manera, en caso de querer ampliar el software, es posible extender la funcionalidad de añadir usuarios o de manipular directamente la información desde el propio programa, sin necesidad de que el usuario abra los archivos de texto `.cfg` correspondientes para editarlos. Por lo tanto, estrictamente hablando, la única funcionalidad que se utiliza desde el software es la de leer los ficheros de configuración `user_list.cfg`, `car_list.cfg` y `pid_list.cfg`, a través de los métodos `get_users()`, `get_cars()` y `get_pids()` de `cfgparser.py`

Estructura del archivo `user_list.cfg`

```
[user1]
name = Alfredo_Ferreras
birth_date = 16/03/1994
license_year = 2018
total_km = 500
n_session = 0

[user2]
name = Senior_Rezungon
birth_date = 23/05/1974
license_year = 1992
total_km = 85000
n_session = 2

[user3]
name = Seniora_Conductora
birth_date = 8/01/1961
license_year = 1983
total_km = 175000
n_session = 1
```

Figura 5.2 Estructura del archivo `user_list.cfg`

La estructura de este archivo, tal y como puede verse en la figura 5.2, consta de una sección por usuario, identificado mediante `userx`, donde `x` es el número de usuario.

Cada sección contiene cinco campos de relevancia que diferencian a cada usuario del resto: nombre (sin espacios), fecha de nacimiento, año de obtención del permiso de conducción, número total de km recorridos y número de sesiones de captura realizadas.

Estructura del archivo *car_list.cfg*

```
[car1]
maker = obdsim
model = OBdII
gear_box = Automatic
n_gears = *
fuel = gas

[car2]
maker = Renault
model = Zoe
gear_box = Automatic
n_gears = *
fuel = diesel

[car3]
maker = Mercedes
model = Class E
gear_box = Standard
n_gears = 6
fuel = diesel
```

Figura 5.3 Estructura del archivo *car_list.cfg*

La estructura de este archivo, tal y como puede verse en la figura 5.3, consta de una sección por coche, identificado mediante *carx*, donde *x* es el número de coche.

De nuevo, cada sección dispone de cinco campos de relevancia que diferencian a cada coche del resto: fabricante, modelo, tipo de caja de cambios (manual o automática), número de marchas (en caso de que sea manual, si tiene sexta o no) y tipo de combustible (diésel, gasolina).

Estructura del archivo *pid_list.cfg*

```
[car1]
010c = RPM
0110 = MAF
0111 = THROTTLE_POS
#~ 0109 = LONG_FUEL_TRIM_2
#~ 0108 = SHORT_FUEL_TRIM_2
010d = SPEED
#~ 0105 = COOLANT_TEMP
0104 = ENGINE_LOAD
#~ 0107 = LONG_FUEL_TRIM_1
#~ 0106 = SHORT_FUEL_TRIM_1
#~ 0139 = O2_S6_WR_CURRENT
#~ 0156 = LONG_O2_TRIM_B1
#~ 0157 = SHORT_O2_TRIM_B2
#~ 0154 = EVAP_VAPOR_PRESSURE_ALT
#~ 0155 = SHORT_O2_TRIM_B1
#~ 0152 = ETHANOL_PERCENT
#~ 0153 = EVAP_VAPOR_PRESSURE_ABS
0150 = MAX_MAF
0151 = FUEL_TYPE
```

Figura 5.4 Estructura del archivo *pid_list.cfg*

La estructura de este archivo, tal y como puede verse en la figura 5.4, consta de una sección por coche, identificado mediante *carx*, donde x es el número de coche.

En este caso, cada coche dispone de una serie de PID que se han obtenido mediante la función “Escanear PID” del menú principal. Al añadir un nuevo coche, la sección debe estar vacía y ejecutarse esta función en primer lugar. Posteriormente se cierra el software y se abre el archivo de configuración, comentando con un # aquellos PID que no se desee leer durante la sesión de captura. Cada línea consta de un número de PID y el nombre o descripción que la librería *obd* de Python le asigna.

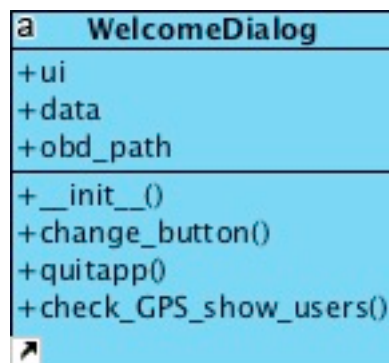


Figura 5.5 Diagrama de clase de *WelcomeDialog*

Por otra parte, tal y como puede observarse en la figura 5.5, en cuanto a la clase `WelcomeDialog`, se trata de una clase derivada de `QDialog` y que permite mostrar una ventana con botones.

Interfaz de Usuario (UI)

En el método `__init__()` se inicializan todos los botones y etiquetas correspondientes a través de la clase `login`, generada de manera automática por `QtDesigner` (`pyuic4`) a partir de un archivo `.ui` (en lenguaje XML) de manera automática. Esto tiene la doble ventaja de separar interfaz de implementación y de automatizar el proceso de diseño de la propia interfaz, dando lugar a un ahorro de tiempo considerable, así como a una mayor legibilidad del código. Además, favorece enormemente la reutilización: si hicieran falta cambios en la parte de interfaz gráfica sólo habría que abrir el archivo `login.ui` con `QtCreator`, realizar las modificaciones oportunas, guardar y volver a generar el archivo `login.py` con la herramienta `pyuic4` (PYTHON User Interface Creator 4). Así, basta con colocar todos los archivos de interfaz en una carpeta separada (UI) del directorio raíz e importar el fichero `.py` según corresponda.

Así, cada Diálogo del software contará, además del método `__init__()`, con un atributo "ui" que almacenará el objeto que incluye a todos los demás objetos de la interfaz gráfica. En caso de que se desee acceder a uno en concreto, basta con referenciarlo mediante la sintaxis estándar de Python. P. ej: `self.ui.pushButtonExit.setText("Saliendo...")` cambia el texto del botón "Salir" a "Saliendo". De manera general, cualquier objeto es accesible mediante `self.ui.nombreDelObjeto`.

Por último, el atributo `obd_path` almacena la ruta en el sistema del dispositivo con el que se establece la comunicación obd. Esta se obtiene como argumento de consola (`argv`) al llamar al lanzador de la aplicación `__main__.pyw`, y se pasa también como argumento al instanciar `LoginDialog()`.

Métodos heredados de QDialog

Al ser `WelcomeDialog` una clase derivada de `QDialog`, hereda todos sus métodos y atributos. Dos de los métodos que siempre se llamarán sobre el propio Diálogo nada más levantar la UI serán `setWindowFlags()` y `setAttribute()`. Los flags de ventana permiten modificar el comportamiento de la misma. Para ver todas las opciones disponibles basta con consultar la documentación de Qt4. Para este software, se busca aprovechar al máximo el tamaño de la pantalla de 2.8 pulgadas, por lo que sería conveniente eliminar las decoraciones de las ventanas. Esto se realiza mediante

```
self.setWindowFlags(QtCore.Qt.FramelessWindowHint),
```

 donde
`Qt.FramelessWindowHint` es el nombre del flag. La documentación de PyQt4 advierte que el resultado puede variar de sistema en sistema pero que la mayoría de gestores de ventanas modernos permiten esta opción. El entorno PIXEL de Raspbian, derivado de LXDE, emplea Openbox como gestor de ventanas, que proporciona soporte para ventanas sin borde por lo que se puede aprovechar la superficie ocupada por el marco de la ventana y la barra de título para uso de la interfaz de usuario.

Un caso especial de métodos heredados, y presente tanto en las clases derivadas de `QDialog` como `QMainWindow` y otras, es el método **connect**. Este método es fundamental, pues es el que activa la funcionalidad de señales y ranuras, un mecanismo de comunicación entre objetos de alto nivel que permite mediante el disparo y la captura de eventos realizar acciones en función de estos.

Por lo tanto, siempre habrá una sección con varias llamadas al método `self.connect()`:

```
self.connect(
    self.ui.pushButtonUserList, QtCore.SIGNAL('pressed()'),
    self.change_button)          #user-made slot
self.connect(
    self.ui.pushButtonUserList, QtCore.SIGNAL('released()'),
    self.check_GPS_show_users)  #user-made slot
self.connect(
    self.ui.pushButtonQuit, QtCore.SIGNAL("clicked()"),
    self.quitapp)                #user-made slot
```

La sintaxis es siempre de la misma manera para métodos creados por el usuario: *connect* (objeto que emite la señal, señal emitida, función definida por el usuario que realiza las acciones correspondientes). Normalmente, la señal emitida está ya predefinida por Qt, como botón presionado, botón liberado o hacer clic). No obstante, se pueden emitir señales específicas definidas por el desarrollador con parámetros incluidos (p.ej. para comunicar un hilo secundario con el principal, como se verá en el ciclo de captura de datos).

En lo que se refiere al método asociado, no es más que un método de clase que el desarrollador añade de manera adicional, y es aquí donde realmente se implementa toda la funcionalidad del programa: en el caso de `self.change_button()` se sombrea el botón de mostrar la lista de usuario

para indicarle que espere al fix del GPS; en el caso de `self.quitapp()` se lanza una excepción `SystemExit` que provoca la salida de la aplicación; por último, en el caso de `self.check_GPS_show_users()` se desactiva el demonio `gpsd` del sistema, se abre un socket `gpsd` bajo demanda junto con un flujo de datos `data_stream` y se vigila por si hay cambios. Se intenta (try) capturar un objeto del flujo de datos. Si no se encuentran objetos nuevos dentro del flujo de datos, se emite una excepción advirtiendo al usuario que verifique el enlace GPS. Si hay un enlace funcional pero no hay fix, entonces el dispositivo GPS se sondeará hasta que se obtenga con éxito.

Atributos heredados de QDialog

En cuanto a los atributos heredados de `QDialog`, pueden cambiarse mediante el método `self.setAttribute()`. Puede verificarse también su valor con el método `self.testAttribute()`. Ambos precisan de argumento una constante de tipo `Qt.WidgetAttribute` (que denota un enum de constantes). Los atributos de la propia clase `QDialog` son heredados por `WelcomeDialog`, además de `ui`, `data` y `obd_path`, que son los propios de dicha clase. De los heredados interesa activar la opción de que Qt elimine el objeto `WelcomeDialog` una vez se cierre el diálogo. De esta manera se busca que Qt gestione la memoria de manera semiautomática, forzando al software a crear los diálogos bajo demanda pero eliminándolos de memoria en cuanto dejen de usarse (es decir, cuando se active el evento de cierre). Esta opción se activa mediante la siguiente sentencia:

```
self.setAttribute(QtCore.Qt.WA_DeleteOnClose, True),
```

donde `Qt.WA_DeleteOnClose` es una constante de tipo `Qt.WidgetAttribute`.

5.3 Diálogo de inicio de sesión

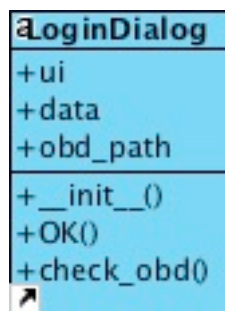


Figura 5.6 Diagrama de clase de `LoginDialog`

El dialogo de inicio de sesión, tal y como puede verse en la figura 5.6, se encarga de leer los datos de usuario y de presentarlos. Se busca que el usuario se identifique para poder clasificar los datos de la sesión. También se aprovecha para hacer una prueba de conexión obd, análoga a la de gps de la clase WelcomeDialog.

Los atributos de la clase LoginDialog, tanto *ui* como *data* y *obd_path*, son los mismos que en la clase anterior.

No obstante, el método `init()`, aparte de inicializar la interfaz de usuario con los cuadros combinados y los dos `pushButton` y establecer los flags y atributos, configura los cuadro combinados con los datos *parseados* en `WelcomeDialog` (y contenidos en *data*). Así, el siguiente código se encarga de leer los usuarios y coches de `UserData` y mostrarlos de manera ordenada mediante un desplegable al usuario del software:

```
for one_user in self.data.user_list: #show registered users
    self.ui.comboBoxUser.addItem(one_user['name'])
for one_car in self.data.car_list: #show added cars
    self.ui.comboBoxVehicle.addItem(one_car['maker']+
    +one_car['model'])
```

Para recorrer los datos que pasan por el objeto hacemos lo siguiente: con una estructura *for element in group_of_elements*: se recorre primero la lista de usuarios guardada en `data` (`data.user_list`) y se añade cada usuario de la lista al cuadro combinado de usuarios. Se procede de manera análoga con los vehículos. Cabe destacar que los objetos tipo `comboBox` de Qt ya poseen un método `addItem` preparado para añadir nuevos elementos de manera secuencial.

Por último, es importante señalar que las tres listas (la de usuarios, la de coches y la de PID) son diccionarios de Python, y como tales, requieren una clave para identificar cada valor. La clave en el caso de un usuario es `'name'`, de manera que lo que se almacena es su nombre real, y en el caso de un vehículo es `'maker'`, y `'model'`, almacenándose el valor de ambos en un único campo.

De esta manera, para cada lista podemos referenciar según sección y/o campo del archivo `.cfg` correspondiente. Esto, aunque pueda parecer al principio un tanto confuso, permite a la larga una mejor reutilización y mantenimiento del código, pues un ser humano siempre será más capaz de

leer sin problemas texto natural que llevar la cuenta de índices acorde a cada campo con números enteros. Puesto que Python es un lenguaje de alto nivel, esto es sencillo y fácilmente implementable, disponiendo de estructuras como los diccionarios que facilitan enormemente el trabajo con lotes de información.

Por último, los métodos `OK()` y `check_obd()` se encargan de implementar funcionalidades específicas que ocurren al presionar y al liberar el botón de OK (Aceptar), respectivamente.

Cuando se pulsa, `OK()` se encarga de sombrearlo e indicarle al usuario que espere al resultado del test de conexión OBD.

```
mb = QtGui.QMessageBox (u'¡Éxito!', unicode(connection.status())+
    u"\nmediante"+unicode(connection.protocol_name()),
    QtGui.QMessageBox.Information, QtGui.QMessageBox.Ok, 0, 0)
mb.setWindowFlags(QtCore.Qt.FramelessWindowHint)
mb.setAttribute(QtCore.Qt.WA_DeleteOnClose, True) # delete dialog
on close
mb.exec_() #prevent focus loss
```

Cuando se libera, `check_obd()` se ejecuta y se establece conexión obd. Hay, además, una sección de código que se encarga de advertir al usuario en caso de mal funcionamiento o de confirmar que todo ha ido bien. Para ello se usa `QMessageBox` de `QtGui`.

```
from MainMenuDialog import MainMenuDialog
    user_index = self.ui.comboBoxUser.currentIndex()
    car_index = self.ui.comboBoxVehicle.currentIndex()
    mainm =
    MainMenuDialog(self.obd_path, self.data, user_index,
    car_index)
    self.close()
```

Antes de instanciar el menú principal, se conserva el índice de usuario y vehículo que se han seleccionado. Para ello se guarda el índice de los cuadros combinados correspondientes, que coinciden con el orden de las listas pues se han añadido de manera secuencial.

5.4 Menú principal

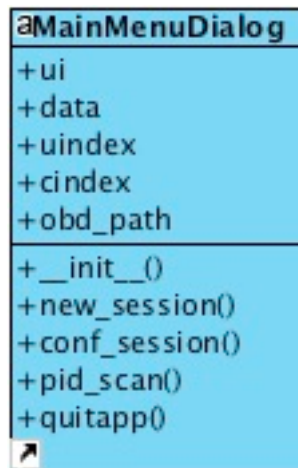


Figura 5.7 Diagrama de clase de `MainMenuDialog`

El menú principal es realmente la primera pantalla del software donde podemos seleccionar y comenzar a utilizar las diferentes funciones.

El código, tal y como puede observarse en el diagrama de clases de la figura 5.7, es similar al de la clase `LoginDialog`: los atributos y métodos ya presentes continúan invariables, con la salvedad de `__init__()`, donde se incluye todo el nuevo código referente a los *signals* y *slots* que corresponden a cada botón: el de nueva sesión, configurar sesión, escanear PID de una nueva lista y salir del software.

```

#----- NEW SESSION -----
self.connect(
    self.ui.pushButtonNewSession,QtCore.SIGNAL('clicked()'),
    self.new_session)
#----- CONF SESSION -----
self.connect(
    self.ui.pushButtonConfSession, QtCore.SIGNAL('clicked()'),
    self.conf_session)
#----- SCAN PIDS -----
self.connect(
    self.ui.pushButtonPidScan, QtCore.SIGNAL('clicked()'),
    sel.pid_scan)
#----- QUIT -----
self.connect(
    self.ui.pushButtonQuit, QtCore.SIGNAL('clicked()'),
    self.quitapp)

```

El método `new_session()` se encarga de iniciar el diálogo de selección de PID para comenzar una nueva sesión de captura. Recibe como parámetros `obd_path`, `data`, `uindex` (usuario que inicia) y `cindex` (coche que usa).

El método `conf_session()` se encarga de volver a iniciar el diálogo de Login, para volver a seleccionar un usuario y coche distintos. Recibe como parámetros `obd_path` y `data`.

El método `pid_scan()` se encarga de borrar de memoria los pids antiguos correspondientes a ese modelo de coche, abrir una nueva conexión obd y leer el archivo de configuración `pid_list.cfg`. Para ese vehículo, una vez establecida la conexión, aparece una lista de PID que la librería `obd` de Python permite leer.

```

self.data.pid_list[self.cindex].clear()
connection = obd.OBD(self.obd_path)
config = ConfigParser.SafeConfigParser()
config.read('data/pid_list.cfg')
car = u'car' + unicode(self.cindex + 1)
for c in connection.supported_commands:
    mp = str(c)
    mode_pid = mp[0:4] #stores pid number
    self.data.pid_list[self.cindex][mode_pid]=c.name
config.set(car, unicode(mode_pid), unicode(c.name))
with open('data/pid_list.cfg', 'r+') as configfile:
    config.write(configfile)

```

Cada PID tiene un nombre y un número, que se almacena en memoria. Este proceso se repite de manera sucesiva con todos los PID soportados. Una vez finalizado el proceso, se escribe el resultado (en formato número = nombre)

en el archivo, sustituyendo en él la sección correspondiente al vehículo en caso de que existiera.

Por último, el método `quitapp()` se encarga de terminar la ejecución del software lanzando una excepción.

5.5 Diálogo de selección de PID

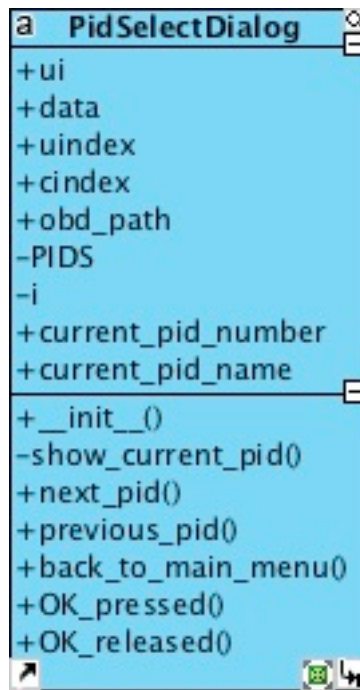


Figura 5.8 Diagrama de clase `PidSelectDialog`

El diálogo de selección de PID muestra al usuario una serie de PID a elegir mediante una lista en modo rueda (es decir, cuando llega al principio sigue por el final y cuando llega al final sigue por el principio) con su nombre y su número. Para este fin, se definen cuatro atributos nuevos, tal y como puede observarse en la figura 5.8: `PIDS`, `i`, `current_pid_number` y `current_pid_name` (siendo los dos primeros privados, es decir, no accesibles desde fuera mediante el operador “.”).

- `PIDS` contiene únicamente los números de PID que se van a mostrar. Para obtener los nombres de PID, también se requiere el número de PID actual.
- `i` no es más que el índice que indica la posición de la lista de `PIDS` mostrada en ese momento. Funciona como contador.
- `current_pid_number` contiene el número del PID que se muestra en ese instante.

- *current_pid_name* contiene el nombre del PID que se muestra en ese instante también.

En cuanto a métodos de clase, `__init__()` adquiere nuevas sentencias para poder definir los nuevos atributos que se acaban de describir:

```
#PID LIST
self.__PIDS = sorted(
    self.data.pid_list[self.cindex],
    key=self.data.pid_list[self.cindex].get)
self.__i = 0
#CURRENT PID
self.current_pid_number = self.__PIDS[self.__i]
self.current_pid_name =
self.data.pid_list[self.cindex][self.__PIDS[self.__i]]
#SHOW CURRENT PID
self.__show_current_pid() #shows first element of the
                          #pid 'wheel' of elements.
```

La nueva lista de PID se obtiene ordenando la del vehículo correspondiente en `pid_list.cfg`, de manera que se muestre correctamente colocada al usuario.

Una vez inicializados todos los nuevos atributos, se emplea el método `show_current_pid()` (también privado y no accesible desde fuera) para cambiar los nombres de las etiquetas de nombre de PID y número de PID,

```
self.current_pid_number = self.__PIDS[self.__i]
self.current_pid_name =
self.data.pid_list[self.cindex][self.__PIDS[self.__i]]
self.ui.labelPIDname.clear()
self.ui.labelPIDnum.setText(self.current_pid_number)
self.ui.labelPIDname.setText(self.current_pid_name)
```

actualizándolas adecuadamente:

Como puede observarse, el código son apenas 5 líneas pero permite separar interfaz (“mostrar el nombre y número de pid actual”) de su implementación (“cambio del texto de las etiquetas de nombre y número de PID”) facilitando enormemente la lectura y comprensión en los otros métodos. De ahí su carácter meramente funcional y no accesible desde el exterior.

En cuanto al resto de métodos, `next_pid()` (figura 5.9) está asociado al símbolo de avance hacia adelante y `previous_pid()` (figura 5.10) al símbolo de avance hacia atrás. Su función es avanzar o retroceder una unidad en la lista

de PID respectivamente y llamar a *show_current_pid()* para que se actualicen los cambios correctamente.

Ambos métodos implementan un mecanismo sencillo de *boundary check* que permite un movimiento tipo “rueda” continuo conforme se alcanzan los límites superiores e inferiores de la lista; en caso de llegar al superior se regresa al inferior y viceversa:

```
def next_pid(self):
    """Increments index number by 1 unit, making sure it is not out
    of range, that is, when end is reached it goes back to the FIRST
    element."""
    self.__i+=1                #increment index
    if self.__i < len(self.__PIDS): #if index is NOT out of range
        self.__show_current_pid() #proceed to show current pid
    else:                       #if index IS out of range
        self.__i=0              #go back to the first element
        self.__show_current_pid() #proceed to show the first element
```

Figura 5.9 Cuerpo de la función *next_pid*

```
def previous_pid(self):
    """Decrements index number by 1 unit, making sure it is not out
    of range, that is, when end is reached it goes back to the LAST
    element."""
    self.__i-=1                #decrement index
    if self.__i >= 0:           #if index is NOT out of range
        self.__show_current_pid() #proceed to show current pid
    else:                       #if index IS out of range
        self.__i=(len(self.__PIDS)-1) #go back to the last element
        self.__show_current_pid() #proceed to show the last element
```

Figura 5.10 Cuerpo de la función *previous_pid*

back_to_main_menu() permite retroceder al diálogo de Menú Principal.

OK_pressed() se ejecuta al presionar el botón de OK e indica al usuario que espere a que se cargue el diálogo que mostrará el valor instantáneo del PID seleccionado.

OK_released() se activa cuando el usuario deja de presionar el botón de OK y se encarga de ejecutar la siguiente pantalla. Como novedad, se pasa un argumento adicional que es el nombre del PID actualmente seleccionado (*self.current_pid_name*).

5.6 Diálogo de muestra de PID

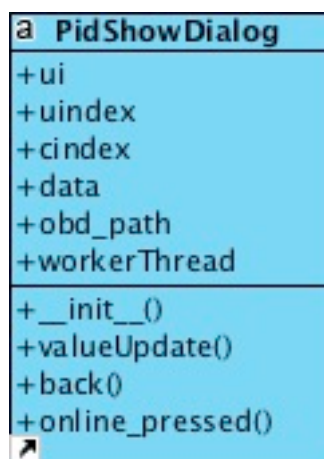


Figura 5.11 Diagrama de clase PidShowDialog.

Esta clase, cuyo diagrama se muestra en la figura 5.11, se encarga de mostrar el valor del PID en tiempo real, y de presentar al usuario la posibilidad de abrir el diálogo de procesado *online* o de volver hacia atrás para mostrar otro PID durante la sesión de captura. Para ello, se observó que resultaba del todo inviable que una única clase asumiera todas las funciones para establecer conexión OBD, comunicarse con el GPS, preguntar cada segundo y recoger los datos en una base de datos SQLite, todo ello sin bloquear la visualización del valor seleccionado en tiempo real (que no queda registrado en la base de datos). Si todo se ejecuta de manera secuencial, se percibe un retardo de unos cientos de ms, lo suficientemente grande como para entorpecer la experiencia de usuario, ya que el método *valueUpdate()* encargado de ello se ejecutaría varios cientos de ms más tarde después de haber obtenido los datos. En el ínterin, la interfaz de usuario del software quedaría bloqueada. Por este motivo, se decidió que una de las variables de esta clase fuera un objeto tipo hilo, que asumiera todas estas funciones en paralelo sin bloquear la interfaz de usuario. Este hilo recibió el nombre de *workerThread*, y es una clase nueva derivada de *QThread*. Un *thread* o hilo no es más que un objeto que, además de los métodos y variables definidos por el usuario, hereda un método *self.start()* que inicia la ejecución en bucle del contenido del método *self.run()* (que es donde debe colocarse la lógica que controla el bucle) junto con un método *self.terminate()* que termina inmediatamente la ejecución del hilo. En la figura 5.12 se presenta un resumen de todos los métodos de la clase *WorkerThread*:

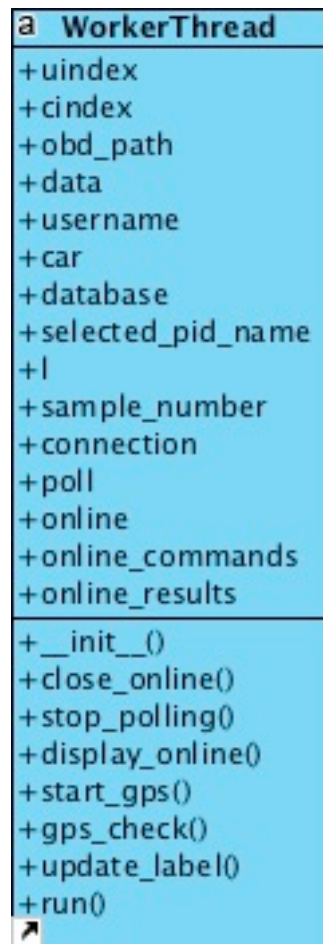


Figura 5.12 Diagrama de la clase WorkerThread.

El objeto WorkerThread, al llamar a su método *start()* en el constructor de PidShowDialog, crea un objeto en *database* encargado de gestionar la escritura de los valores de los PID enviados al vehículo. A continuación, *start_gps()* se encarga de abrir un hilo con la librería *agps*, que permanecerá ejecutándose en segundo plano, recibiendo un flujo de datos constante del GPS. Acto seguido, se crea un objeto *obd* y se almacena en *connection*. Después, se instancia otro objeto tipo hilo encargado del procesamiento de datos *online*. Para terminar, se establece la variable *poll* a True y con ello comienza a ejecutarse en bucle el código contenido dentro del método *run()* del hilo.

Se continúa comprobando los datos del GPS en *gps_check()*, y se añaden a una lista temporal para contenerlos, *l*.

Tras esto, *update_label()* se encarga de enviar al coche el PID cuyo valor va a mostrarse en tiempo real y de obtenerlo como respuesta; también se consulta a continuación el valor de otros PID como RPM, THROTTLE_POS o SPEED y guardarlos en la lista *l*. Serán necesarios para el posterior procesado online.

Para terminar, se pasa la lista de valores recopilados al hilo encargado del procesamiento online y se procede a mostrar el valor del PID que se había seleccionado.

Para ello, es necesario utilizar el mecanismo de señales y slots, de manera que el hilo *workerThread* pueda comunicarse con la clase *PidShowDialog*.

5.7 Ventana de procesamiento online

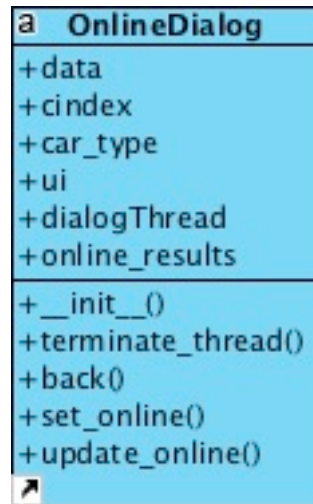


Figura 5.13 Diagrama de la clase *OnlineDialog*.

El diagrama de clase de la figura 5.13 muestra el diálogo de procesamiento online. A él se accede tras presionar el botón de *online* en la interfaz de usuario. Muestra una pantalla con varias estadísticas sobre el estado de la conducción, buscando proporcionar datos para optimizarla y conseguir, entre otros objetivos, un consumo más eficiente de combustible o, en el caso de los eléctricos, una conducción más suave y respetuosa con el entorno que alargue la duración de la vida de las baterías.

Respecto al código, es análogo al de la clase anterior; se ha decidido crear un hilo para realizar las operaciones de procesamiento por separado, de manera que no se bloquee la ejecución del programa principal durante la realización de los cálculos.

Tras abrirse el diálogo de procesamiento online, se inicia el hilo encargado de las operaciones en sí. Las variables y métodos del hilo de procesamiento pueden consultarse en la figura 5.14.



Figura 5.14 Diagrama de la clase DialogThread.

Destacan aquí una serie de variables encargadas de almacenar los *warnings* (w, de ahí la primera parte de su nombre) o avisos al usuario, que son los que se mostrarán por pantalla. También, en *w_history* se almacenan los valores anteriores para realizar, por ejemplo, la desviación típica respecto de ellos. Por último, la variable *thread_running* es una variable de control, encargada de comenzar los cálculos de procesado en tiempo real en caso de que su valor sea True.

También se dispone de un método *back()* encargado de esconder (que no cerrar o detener) la ventana de procesado online para regresar a la pantalla anterior. Puesto que todo se ejecuta mediante hilos, estos están siempre corriendo y haciendo los cálculos correspondientes en segundo plano, hasta que sus variables de control (*thread_running* en caso del procesado online y *poll* en caso de mostrar únicamente el valor de un PID por pantalla).

Capítulo 6. Presupuesto

Todos los materiales utilizados en este TFG son relativamente económicos y de fácil adquisición. En el desglose de gastos materiales no se incluye el software, puesto que es gratuito y de libre acceso en Internet.

Ítem	Precio
Conector OBD	49,95 €
Ratón Bluetooth	42.20 €
Teclado Bluetooth	39.99 €
Módulo GPS	39,95 €
Raspberry Pi	39.95 €
Adafruit PiTFT Display	34,95 €
Cable TTL Serie - USB	9,99
Conector WiFi	4 €
Módulo Bluetooth	5 €
TOTAL	143,84 €

Se presenta a continuación también un desglose de los gastos referidos al salario de un ingeniero júnior, con un salario medio de 10,50 €/h y 475 horas de trabajo dedicadas (sin incluir el periodo de prácticas):

Ítem	Precio
Gastos de personal	4987,5 €

En total, los gastos de materiales y laborales de este TFG ascienden a un total de: 5131,34€.

Capítulo 7. Conclusiones y líneas futuras

En este documento, he intentado exponer de manera sucinta y fiel casi 7 meses de trabajo continuo para el desarrollo de una aplicación que se ajuste a los requisitos planteados de manera conjunta con mi tutor. Durante el proceso, me he encontrado con problemas tales como falta de documentación, rotura (por dos veces) de la pantalla PiTFT y, sobre todo, la dificultad de encontrar información sobre OBD y cómo obtener los datos de un coche, no ya eléctrico, sino meramente de combustión. Dificultades, todas ellas, presentes en el mundo profesional.

El principal objetivo del TFG ha sido aprender e incrementar mis conocimientos técnicos uniendo tanto hardware como software, que están íntimamente relacionados y que en muchos casos deben trabajar de manera conjunta y coordinada para tareas críticas. Gracias a todos estos meses de trabajo, he podido obtener mi primera oportunidad profesional en el ámbito de los sistemas embarcados, donde se ha valorado muy positivamente las capacidades adquiridas al desarrollar este TFG. Cabe mencionar, por ejemplo, el uso de hilos para realizar varias tareas de manera simultánea sin bloquear el software y dentro de un hardware limitado. O la necesidad de investigar alternativas al conexionado de GPS por puerto serie al estar los puertos GPIO ocupados por la pantalla, lo cual obligó a desarrollar una alternativa con un cable conversor USB a serie TTL. Todas estas situaciones pueden darse en cualquier momento y se ha de tener recursos para solucionarlas.

Se ha desarrollado, en definitiva, una aplicación que dispone de un sistema de seguimiento de los datos de los usuarios del software, que clasifica sus sesiones en directorios de manera ordenada, que guarda la información de las sesiones de conducción en registros y que es capaz de cargar las preferencias de cada usuario desde archivos de configuración, que emplea tecnologías modernas y propias de sistemas embarcados (como Linux empotrado, Qt o SQLite), que presenta una interfaz gráfica agradable y preparada para su uso como pantalla táctil, que proporciona al usuario la capacidad de visualizar el

valor de un PID en tiempo real, así como datos de eficiencia en la conducción. Todo ello de manera fluida y fácilmente escalable gracias a un diseño modular y compacto.

Durante todo este tiempo, y más a título personal, me he dado cuenta del ingente esfuerzo que supone desarrollar un sistema nuevo casi completamente desde cero. Pese a partir de tecnologías probadas ya existentes, y de contar con la ayuda de mi tutor, este TFG ha tenido una parte previa de investigación y documentación importantísima. Esto muchas veces no es visible en el resultado final, no solo de este trabajo sino de cualquier producto y es una lección importante a tener en cuenta de cara a futuros desarrollos.

El resultado final es una aplicación prototipo o boceto pero funcional en esencia: cumple con el objetivo fijado de facilitar el diagnóstico y proporciona una base de código sobre la que trabajar para su ampliación en futuros proyectos.

Por último, me gustaría destacar algunas líneas de trabajo futuras. Entre ellas, introducir las mejoras mencionadas en el capítulo 6 para mejorar el desempeño en todo tipo de coches. A nivel software, cabe mencionar las siguientes ampliaciones: disponer de un sistema de registro de usuarios más sofisticado, hacer compatible el software con el máximo número de coches eléctricos posible o mejorar de manera sensible la interfaz gráfica, añadiendo nuevos menús y la posibilidad de seleccionar más comprobaciones de manera individual mediante un sistema de casillas de verificación. También podría desarrollarse una interfaz para permitir a los usuarios cambiar sus datos y los PID que desean consultar. Para ello, sería necesario una pantalla más grande, de al menos 5 pulgadas. También, y en línea con mejoras hardware adicionales, se podría añadir una antena externa al GPS para mejorar su cobertura y la velocidad de *fix* e incluso diseñar una nueva carcasa mediante una impresora 3D que también tuviera espacio adicional para un altavoz sencillo y una pequeña batería. Esto permitiría realizar sesiones más autónomas sin la necesidad de emplear cableado externo. Puesto que el futuro pasa por la conectividad, para acabar me gustaría plantear la posibilidad de incorporar conectividad LTE para poder compartir todos los datos en tiempo real.

Bibliografía

- [1] (s.f.). Recuperado el 9 de Julio de 2018, de Effbot (Python FAQ): <http://effbot.org/pyfaq/can-python-be-compiled-to-machine-code-c-or-some-other-language.htm>
- [2] Adafruit. (17 de Agosto de 2018). *Adafruit - PiTFT user manual*. Obtenido de <https://cdn-learn.adafruit.com/downloads/pdf/adafruit-pitft-28-inch-resistive-touchscreen-display-raspberry-pi.pdf>
- [3] Barreto, V. (17 de Agosto de 2017). *GEOTAB*. Recuperado el 24 de 2 de 2018, de <https://www.geotab.com/blog/obd-ii/>
- [4] Brendan, W. (2016). *Python-obd*. Recuperado el 12 de Julio de 2018, de <https://python-obd.readthedocs.io/>
- [5] Company, T. Q. (7 de Marzo de 2018). *Qt Wiki*. Recuperado el 12 de Julio de 2018, de https://wiki.qt.io/About_Qt/es
- [6] Coria, D. A. (Febrero de 2014). *UNAM*. Obtenido de <http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/3407/Tesis.pdf?sequence=1>
- [7] Corinne, T. (24 de Febrero de 2018). *sparkfun*. Recuperado el 24 de 2 de 2018, de <https://learn.sparkfun.com/tutorials/getting-started-with-obd-ii>
- [8] DePriest, D. (2012). *GPS informatio - NMEA data*. Recuperado el 20 de Julio de 2018, de <http://www.gpsinformation.org/dale/nmea.htm>
- [9] Fried, L. (2016). *Adafruit - Ultimate GPS*. Recuperado el 19 de Junio de 2018, de <https://cdn-learn.adafruit.com/downloads/pdf/adafruit-ultimate-gps.pdf>
- [10] Kieras, D. (Junio de 2016). *Using C++11's Smart Pointers*. Recuperado el 9 de Julio de 2018, de http://www.umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf
- [11] Murphy, E. (2012, Abril 4). Retrieved from <https://www.quora.com/What-are-the-advantages-of-Python-over-C++>
- [12] Novikov, V. (24 de Febrero de 2018). Recuperado el 24 de 2 de 2018, de <http://science.donntu.edu.ua/ks/novikov/library/files/3.pdf>
- [13] *Outils OBD Facile*. (Agosto de 2018). Recuperado el 12 de Agosto de 2018, de <https://www.outilsobdfacile.com/obd-mode-pid.php>
- [14] PSF. (2018). Recuperado el 11 de Julio de 2018, de The Python Standard Library Documentation: <https://docs.python.org/3/library/tk.html>
- [15] S. Raymond, K. M. (s.f.). *catb*. Recuperado el 14 de Julio de 2018, de <http://catb.org/gpsd/gpsd.html>

- [16] Sam. (24 de Septiembre de 2011). *Stackoverflow*. Recuperado el 11 de Julio de 2018, de <https://stackoverflow.com/questions/7539401/gui-layout-using-tk-grid-geometry-manager>
- [17] *Slant*. (2016). Recuperado el 12 de Julio de 2018, de https://www.slant.co/versus/16724/22768/~tkinter_vs_pyqt
- [18] SQLite Tutorial. (2016). *What is SQLite*. Recuperado el 2018, de <http://www.sqlitetutorial.net/what-is-sqlite/>
- [19] Tseng Chien-Ming, W. Z. (2016). Data Extraction from Electric Vehicles through OBD and Application of Carbon Footprint Evaluation. (U. Masdar Institute of Science and Technology, Ed.) 6.