

Proyecto Fin de Grado  
Grado en Ingeniería Electrónica, Robótica y  
Mecatrónica.

Desarrollo de soluciones IoT con Mbed

Autor: José Manuel Salvador Vázquez.

Tutor: Ramón González Carvajal.

Departamento de Ingeniería Electrónica  
Escuela Técnica Superior de Ingeniería  
Universidad de Sevilla

Sevilla, 2019





Proyecto Fin de Grado  
Grado en Ingeniería Electrónica, Robótica y Mecatrónica.

# **Desarrollo de soluciones IoT con Mbed**

Autor:

José Manuel Salvador Vázquez

Tutor:

Ramón González Carvajal

Catedrático

Departamento de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2019



Proyecto Fin de Grado: Desarrollo de soluciones IoT con Mbed

Autor: José Manuel Salvador Vázquez

Tutor: Ramón González Carvajal

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal



*A mis padres*

*A mis amigos*

*A mis maestros*





# Agradecimientos

---

Son muchas las personas gracias a las cuales sigo donde estoy y que, cada una a su manera, me han ayudado a llegar hasta aquí, a todas ellas gracias.

Gracias a mi familia, mis padres que siempre han estado ahí para apoyarme con todo lo que necesitara para estar mejor durante el camino que ha sido este grado y que, aunque a veces no nos entendamos, siempre me han dado ánimos para seguir con esto y para que confiará más en mis capacidades, y gracias a mis abuelas que, como toda abuela, siempre les da la mayor de las alegrías que su nieto pueda estudiar y sacar adelante un grado universitario.

Gracias a mis amigos que siempre estaban disponibles tras ese examen que me dejaba K.O para animarme el día y que me sacaban de casa tras un largo día de estudio, aunque solo fuera para que me despejase.

Y gracias a mis maestros, esos profesores que incluso sin saberlo hacían que te interesaras por cosas que ni siquiera sabías que te gustaban o que incluso odiabas antes de que ellos te cambiarán de idea. Y de todos los profesores que han pasado por mi vida estos años gracias en especial a mi tutor, Ramón, que me ha hecho ver la ingeniería desde otro punto de vista más real y que, aunque él diga lo contrario, es un gran profesor.

*José Manuel Salvador Vázquez*

*Grado en Ingeniería Electrónica, Robótica y Mecatrónica*

*Sevilla, 2019*



# Resumen

---

El Internet de las cosas o IoT está cobrando cada vez más importancia y es palpable viendo la cantidad de dispositivos inteligentes y conectados a la red que existen hoy en día, dispositivos que van desde un smartphone a un automóvil, pasando por una lavadora.

Esta conexión de objetos cotidianos a internet está mejorando y automatizando procesos que antes realizábamos de manera regular o que ni siquiera teníamos la posibilidad de realizar, como por ejemplo revisar el frigorífico para ver que es necesario comprar o saber en todo momento donde se encuentra un paquete que nos han enviado.

Este proyecto tendrá como objetivo la exploración de Mbed como sistema operativo en tiempo real en microcontroladores basados en ARM para estas aplicaciones IoT y como ejemplo tomaremos un sistema de adquisición de vibraciones con acelerómetro a través de bus CAN para un automóvil.



# Abstract

---

The Internet of Things or IoT is gaining more and more importance and it is remarkable to see the number of smart and connected to the network devices that exists today, these devices go from a smartphone to a car going through a washing machine.

This connection of everyday objects to the Internet is improving and automating the processes that we previously realized in a regular way or not even had the possibility to perform, such as reviewing the refrigerator to see what we have to buy or know at all times where we will find a package sent to us.

The objective of this project will be the exploration of Mbed as a real-time operating system in ARM-based microcontrollers for IoT applications and as an example, we will take a vibration acquisition system with accelerometer via CAN bus for a car.



<b>Agradecimientos</b>	<b>ixx</b>
<b>Resumen</b>	<b>xi</b>
<b>Abstract</b>	<b>xiii</b>
<b>Índice</b>	<b>xv xv</b>
<b>Índice de Figuras</b>	<b>xvii</b>
<b>1 Introducción</b>	<b>1</b>
<b>2 Hardware utilizado</b>	<b>3</b>
2.1. <i>Placa de desarrollo LPCXpresso54608</i>	3
2.1.1 Información general	3
2.1.2 Características del MCU	3
2.1.3 Características de la placa	3
2.1.4 Pinout y layout de la placa	4
2.1.5 Configuración de la placa para su uso con Mbed	6
2.2. <i>Etapas de expansión para la comunicación por bus CAN</i>	7
2.2.1 CAN-BUS Shield V1.2 de DIGITAL	8
2.2.2 Placa de desarrollo NUCLEO-L4R5ZI-P	10
<b>3 Mbed, el rtos de código abierto para el desarrollo de aplicaciones IoT</b>	<b>15</b>
3.1. <i>Descripción general</i>	15
3.1.1. Arquitectura y funcionalidades	15
3.1.1.1. Conectividad	16
3.1.1.2. Seguridad	16
3.1.1.3. Actualización remota de firmware	16
3.1.1.4. Hardware	17
3.1.1.5. Herramientas	17
3.1.1.6. Documentación	17
3.1.2. Interfaces de Programación de Aplicaciones o APIs	17
3.1.2.1. APIs de plataforma	17
3.1.2.2. APIs de drivers	18
3.1.2.3. APIs de RTOS	18
3.1.2.4. APIs de USB	19
3.1.2.5. APIs de sockets de red	19
3.1.2.6. APIs de interfaces de red	19
3.1.2.7. APIs de almacenamiento	19
3.1.2.8. APIs de Bluetooth, LoRaWAN, NFC y seguridad	20
3.2. <i>Entorno de Desarrollo Integrado o IDE</i>	20
3.2.1. Vistazo a Arm Mbed Online Compiler	20
3.2.2. Ejemplo práctico de uso	23
<b>4 Desarrollo del programa objetivo</b>	<b>27</b>
4.1. <i>Exploración de las aplicaciones</i>	27
4.1.1. Comunicación puerto serie asíncrona	27
4.1.2. Temporizadores y "tickers"	30

4.1.3.	Comunicación con protocolo SPI	32
4.1.4.	Conexión a un servidor TCP	35
4.1.5.	Conexión a un servidor UDP	37
4.1.6.	Lectura de un acelerómetro	39
4.1.7.	Comunicación por medio de un bus CAN	41
4.1.8.	Almacenamiento de datos en una tarjeta SD	43
4.2.	<i>Acoplamiento de funcionalidades</i>	44
4.2.1.	Lectura de acelerómetro y envío de datos a un servidor TCP	44
4.2.2.	Lectura de acelerómetro y envío de datos a un servidor UDP	48
4.2.3.	Comunicación CAN - SPI entre dos dispositivos	50
4.3.	<i>Sistema final</i>	59
4.4.	<i>Compatibilidad de las aplicaciones desarrolladas</i>	63
4.4.1.	Prueba de entorno de programación	63
4.4.2.	Comunicación puerto serie	64
4.4.3.	Temporizadores y "tickers"	65
<b>5</b>	<b>Conclusiones</b>	<b>67</b>
	<b>Glosario</b>	<b>69</b>



# ÍNDICE DE FIGURAS

---

Figura 1. Layout de la placa LPCXpresso54608 de NXP	4
Figura 2. Fotografía de la placa LPCXpresso54608 de NXP	5
Figura 3. Pinout de la placa LPCXpresso54608 de NXP (1)	5
Figura 4. Pinout de la placa LPCXpresso54608 de NXP (2)	6
Figura 5. Captura de pantalla de la placa actuando como dispositivo de almacenamiento USB	7
Figura 6. Partes de CAN-BUS Shield V1.2 de DIGITAL	8
Figura 7. Pinout de CAN-BUS Shield V1.2 de DIGITAL	9
Figura 8. Pinout de MCP2515	10
Figura 9. Pinout de MCP2551	10
Figura 10. Placa de desarrollo NUCLEO L4R5ZI-P	11
Figura 11. Pinout de las hileras CN8 y CN9	13
Figura 12. Pinout de las hileras CN7 y CN10	13
Figura 13. Arquitectura básica de una placa Mbed	16
Figura 14. Entorno de programación Arm Mbed Online Compiler	21
Figura 15. Barra de herramientas (Parte 1)	21
Figura 16. Barra de herramientas (Parte 2)	22
Figura 17. Ventana de selección de plataforma	22
Figura 18. Program Workspace	22
Figura 19. Ventana principal del IDE mostrando los archivos internos de un programa	23
Figura 20. El PC detecta la placa de desarrollo como almacenamiento ampliado	24
Figura 21. Programa PruebaEntorno en Arm Mbed Online Compiler	24
Figura 22. Código de PruebaEntorno	25
Figura 23. Referencia a la clase <i>DigitalOut</i> de la librería mbed-os	25
Figura 24. Funciones de la clase <i>Serial</i>	28
Figura 25. Funciones protegidas de la clase <i>Serial</i>	28
Figura 26. Código del programa de prueba para la clase <i>Serial</i>	29
Figura 27. Funcionamiento del programa en <i>Putty</i>	30
Figura 28. Funciones de la clase <i>Timer</i>	30
Figura 29. Funciones de la clase <i>Ticker</i>	31
Figura 30. Código del programa de prueba para las clases <i>Timer</i> y <i>Ticker</i>	31
Figura 31. Funcionamiento del programa en <i>Putty</i>	32
Figura 32. Esquema de protocolo SPI	32
Figura 33. Funciones de la clase <i>SPI</i>	33
Figura 34. Funciones de la clase <i>SPISlave</i>	34
Figura 35. Código del programa de prueba para la clase <i>SPI</i>	34
Figura 36. Respuesta del programa en <i>Putty</i>	34

Figura 37. Funciones de la clase <i>EthernetInterface</i>	35
Figura 38. Funciones de la clase <i>TCPSocket</i>	35
Figura 39. Código de la aplicación para conexión a un servidor TCP mediante ethernet	36
Figura 40. Respuesta del programa por <i>Putty</i>	36
Figura 41. Mensaje que recibe el servidor	37
Figura 42. Mensaje que tiene programado el servidor para responder	37
Figura 43. Funciones de la clase <i>UDPSocket</i>	37
Figura 44. Código del programa de prueba para conexión con un servidor UDP	38
Figura 45. Respuesta del programa en <i>Putty</i>	38
Figura 46. Mensaje recibido por el servidor UDP	38
Figura 47. Mensaje con el que responde el servidor	38
Figura 48. Lista de funciones de la clase <i>MMA8652</i>	39
Figura 49. Código del programa de prueba para lectura del acelerómetro MMA8652	40
Figura 50. Lecturas del acelerómetro cuando la placa está en horizontal	40
Figura 51. Lecturas del acelerómetro cuando giramos la placa hacia la derecha	41
Figura 52. Lecturas del acelerómetro cuando giramos la placa hacia la izquierda	41
Figura 53. Funciones de la clase <i>CANMessage</i>	41
Figura 54. Funciones de la clase <i>CAN</i>	42
Figura 55. Funciones de la clase <i>SDBlockDevice</i>	43
Figura 56. Código de aplicación para envío de datos del acelerómetro a un servidor TCP cada 5 segundos	45
Figura 57. Sistema inicializado y esperando el carácter 'a' para empezar a transmitir	45
Figura 58. Sistema transmitiendo al servidor TCP	46
Figura 59. Tres primeros mensajes recibidos por el servidor TCP	46
Figura 60. Cuarto y quinto mensajes recibidos por el servidor TCP	46
Figura 61. Código de la aplicación para envió manual de datos de un acelerómetro a un servidor TCP	47
Figura 62. Código de aplicación para envío de datos del acelerómetro a un servidor UDP cada 5 segundos	48
Figura 63. Sistema inicializado y esperando el carácter 'a' para empezar a transmitir	49
Figura 64. Sistema transmitiendo al servidor UDP	49
Figura 65. Tres primeros mensajes recibidos por el servidor UDP	50
Figura 66. Cuarto y quinto mensajes recibidos por el servidor UDP	50
Figura 67. Esquema del montaje para comunicación CAN	51
Figura 68. Esquema de bus CAN estándar	51
Figura 69. Niveles de tensión en bus CAN	52
Figura 70. Esquema de la primera parte del montaje para la comunicación CAN	53
Figura 71. Montaje del circuito de prueba	53
Figura 72. Medida de impedancia	54
Figura 73. Medida de tensión	54

Figura 74. Código para el envío de mensaje CAN	54
Figura 75. Lectura de CAN H y CAN L mientras no se envía mensaje	55
Figura 76. Lectura de CAN H y CAN L mientras se envía un carácter	55
Figura 77. Funciones de la clase <i>SEED_CAN</i>	56
Figura 78. Funciones de la clase <i>SEED_CANMessage</i>	56
Figura 79. Código para la recepción del mensaje CAN	57
Figura 80. Montaje para comunicación CAN completa	57
Figura 81. Vista de la información transmitida por el puerto serie de cada una de las placas (Inicio)	57
Figura 82. Vista de la información transmitida por puerto serie de cada una de las placas (Mensajes enviados)	58
Figura 83. Esquema del sistema final	59
Figura 84. Código para el envío de mensaje CAN (Cargado en NUCLEO-L4R5ZI-P)	60
Figura 85. Código para la recepción de mensajes CAN y subida de información a servidor TCP (Cargado en LPCXpresso54608)	60
Figura 86. Montaje del sistema final	61
Figura 87. Vista de la información transmitida por puerto serie de la placa NUCLEO (Envío)	61
Figura 88. Vista de la información transmitida por puerto serie de la placa LPCXpresso (Recepción y subida a servidor)	62
Figura 89. Mensajes recibidos por el servidor	62
Figura 90. Conversión de hexadecimal a texto ASCII de los mensajes recibidos en el servidor	62
Figura 91. Código de PruebaEntorno para NUCLEO-L4R5ZI-P	63
Figura 92. Código para comunicación por puerto serie en NUCLEO-L4R5ZI-P	64
Figura 93. Programa ejecutado y a la espera de recibir algo por el puerto serie	64
Figura 94. Respuesta del programa ante los caracteres introducidos	65
Figura 95. Código del programa de prueba para las clases <i>Timer</i> y <i>Ticker</i> en la NUCLEO-L4R5ZI-P	66
Figura 96. Respuesta de la aplicación por puerto serie al pulsar el botón de usuario	66



# 1 INTRODUCCIÓN

---

*Las cosas no se hacen siguiendo caminos distintos para que no sean iguales, sino para que sean mejores.*

*- Elon Musk -*

**E**n este apartado se desarrollarán las ideas básicas del proyecto y el porqué de la investigación en este campo dando así una idea inicial de lo que se va a ahondar en los siguientes capítulos.

Se puede observar que desde hace unos años el número de dispositivos conectados a internet está aumentando enormemente y que muchos de estos dispositivos son objetos cotidianos que usamos en el día a día como un teléfono móvil, un frigorífico o un coche. Este concepto que estamos atacando tiene el nombre de internet de las cosas (IoT) y se trata de una interconexión digital de objetos cotidianos con internet.

El estudio de este campo es muy interesante ya que se prevé que este aumento de dispositivos IoT no va a descender por el momento sino todo lo contrario y podrán proporcionar funcionalidades como: unas zapatillas que lean y guarden en la nube la actividad física que hemos realizado, un frigorífico que nos avise de la fecha de caducidad de los productos de su interior y sea capaz de realizar una compra, un sistema de marcado que sepa la posición exacta de un producto dentro de un almacén cualquiera de una empresa con almacenes por todo el mundo o un sistema que tome y suba a la nube la posición de un vehículo y su velocidad.

Para el desarrollo de estos dispositivos IoT es necesaria una plataforma de desarrollo que permita diseñar e implementar los programas necesarios para ello. En este proyecto se va a utilizar Mbed como plataforma de desarrollo y sistema operativo del dispositivo utilizado, Mbed es muy interesante como plataforma de desarrollo ya que cuenta con un compilador en línea, permite programar en C++ y es compatible con multitud de dispositivos basados en ARM, esto permite que un mismo programa realizado en Mbed pueda funcionar correctamente en diferentes dispositivos simplemente cambiando las direcciones de entrada y salida. Además, Mbed nos da la ventaja de poder programar en alto nivel y evitar estar aprendiendo el funcionamiento específico del entorno de desarrollo de cada fabricante para usar sus productos.

En teoría Mbed es una buena opción como sistema operativo y plataforma de desarrollo también debido a es una plataforma de código abierto, y por lo tanto gratuita, y a que siempre va a estar disponible ya que la empresa que lo desarrolló es ARM directamente y al ser una empresa que en realidad vive del hardware, vamos a tener acceso a la plataforma durante mucho tiempo, con bastante seguridad.

Viendo la infinidad de aplicaciones que tienen los dispositivos IoT debemos elegir una para realizar y la elegida va a ser la lectura de un acelerómetro a través de un bus CAN y la subida de estos datos a un servidor, este proceso tiene una aplicación directa en automóviles ya que un bus CAN es lo más adecuado para ese entorno debido a que con solo dos cables podemos montar una vía de comunicación a la que conectar muchos sensores y actuadores de manera que el acelerómetro sea solo un ejemplo de todos los que se podrían conectar, además ofrece una alta inmunidad a las interferencias y habilidad de autodiagnóstico.

Además, la aplicación en automóviles es muy interesante debido a la demanda de automatización de estos y el objetivo actual de conseguir que sean más seguros e incluso autónomos.

Para realizar este proyecto se necesita un hardware donde implementar y probar lo que se realice y el sistema

elegido para esto es la placa de desarrollo de NXP LPCXpresso54608 que cuenta con lo necesario para la realización del proyecto y se le añadirá un módulo de expansión para permitir la comunicación por bus CAN, también se implementará parte de proyecto sobre la placa de desarrollo NUCLEO-L4R5ZI-P.

# 2 HARDWARE UTILIZADO

---

El hardware que se va a utilizar para implementar el proyecto es la placa de desarrollo LPCXpresso54608 de NXP y, en principio, el módulo de expansión para bus CAN OM13099, aunque se dispone de más material si hiciera falta, como la placa de desarrollo NUCLEO-L4R5ZI-P, por ejemplo.

## 2.1. Placa de desarrollo LPCXpresso54608 de NXP

### 2.1.1 Información general

La LPCXpresso54608 se trata de una placa de desarrollo fabricada por la empresa holandesa NXP que tiene como microcontrolador un ARM Cortex-M4, por lo que es compatible con el entorno de desarrollo Mbed, y cuenta con numerosos periféricos con los que realizar infinidad de aplicaciones.

### 2.1.2 Características del MCU

La familia de MCU LPC546xx combina la eficiencia energética del núcleo Arm Cortex-M4 a una velocidad de hasta 220 MHz con múltiples opciones de conectividad de alta velocidad, temporizadores avanzados y funciones analógicas. Las capacidades DSP permiten que los dispositivos MCU LPC546xx admitan algoritmos complejos en aplicaciones de uso intensivo de datos. Esta familia ofrece la capacidad de adaptarse a medida que cambian los requisitos ya que proporciona la flexibilidad de 512KB de memoria flash e interfaces de memoria externa, además las opciones de flash admiten configuraciones de memoria interna y externa grandes y flexibles.

Las especificaciones del MCU son:

- LPC54608ET512 Arm Cortex-M4
- 512 KB Flash
- 200 KB SRAM
- Configuraciones flexibles de memoria interna y externa

### 2.1.3 Características de la placa

La placa está compuesta por un dispositivo LPC54608 objetivo con una sonda de depuración compatible con CMSIS-DAP / SEGGER J-Link integrada. La sonda integrada es compatible con MCUXpresso IDE y otras cadenas de herramientas líderes, como las de Keil e IAR. La placa también está equipada con un cabezal estándar de 10 pines que permite el uso de sondas de depuración de terceros.

Además de las características estándar de LPCXpresso V3, esta placa incluye un conjunto completo de interfaces periféricas para permitir a los desarrolladores explorar completamente las capacidades de los dispositivos LPC5460x.

Los periféricos de los que dispone la placa son:

- Pantalla LCD a color de 272x480 con pantalla táctil capacitiva.
- Sonda de depuración Link2 USB integrada de alta velocidad con opciones de protocolo CMSIS-DAP y SEGGER J-Link.
- Los puertos UART y SPI unidos desde el destino LPC546xx al USB a través de la sonda de depuración integrada.
- Soporte para sonda de depuración externa.

- 3 x LED de usuario, además de Reset, ISP (3) y botones de usuario.
- Múltiples opciones de expansión, incluyendo Arduino UNO y PMod.
- Medida de consumo de energía incorporada para MCU LPC546xx objetivo.
- 128 Mb Micron MT25QL128 Quad-SPI flash.
- 16 MB Micron MT48LC8M16A2B4 SDRAM.
- Micrófono digital Knowles SPH0641LM4H.
- Ranura de tarjeta SD / MMC de tamaño completo.
- Acelerómetro NXP MMA8652FCR1.
- Codec de audio estéreo con entrada/salida de línea.
- Puertos USB de alta y máxima velocidad con conector micro A/B para funcionalidad de host o dispositivo.
- Ethernet 10/100 Mbps (conector RJ45).
- Circuito integrado de interfaz de depuración y programación con interfaz USB de alta velocidad [USB MSC].
- Programación Drag-n-drop [USB CDC].
- Puerto serie USB [USB HID].
- CMSIS-DAPon-board.

#### 2.1.4 Pinout y layout de la placa

A continuación, se detalla la distribución de entradas y salidas de la placa:

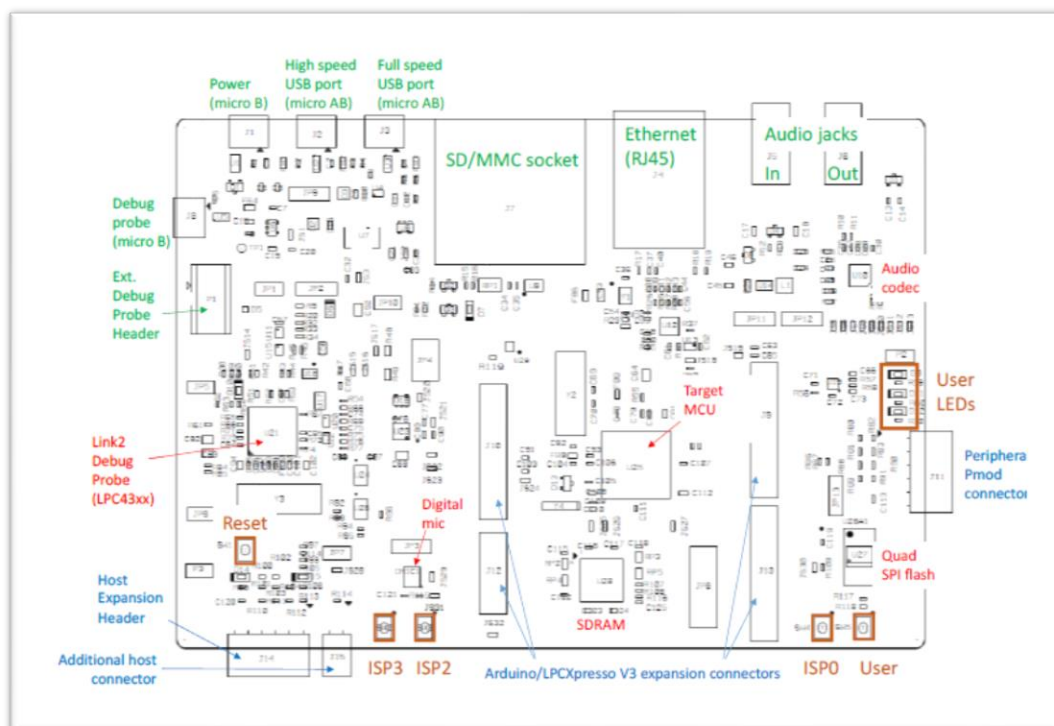


Figura 1. Layout de la placa LPCXpresso54608 de NXP



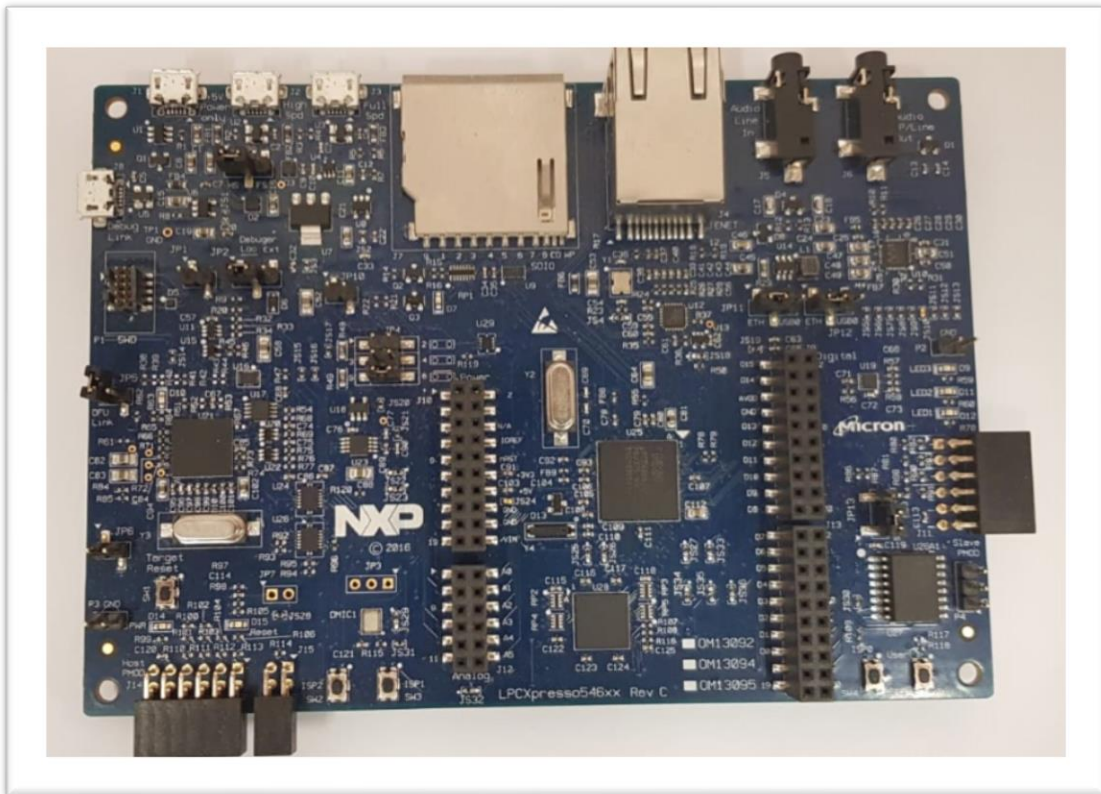


Figura 2. Fotografía de la placa LPCXpresso54608 de NXP

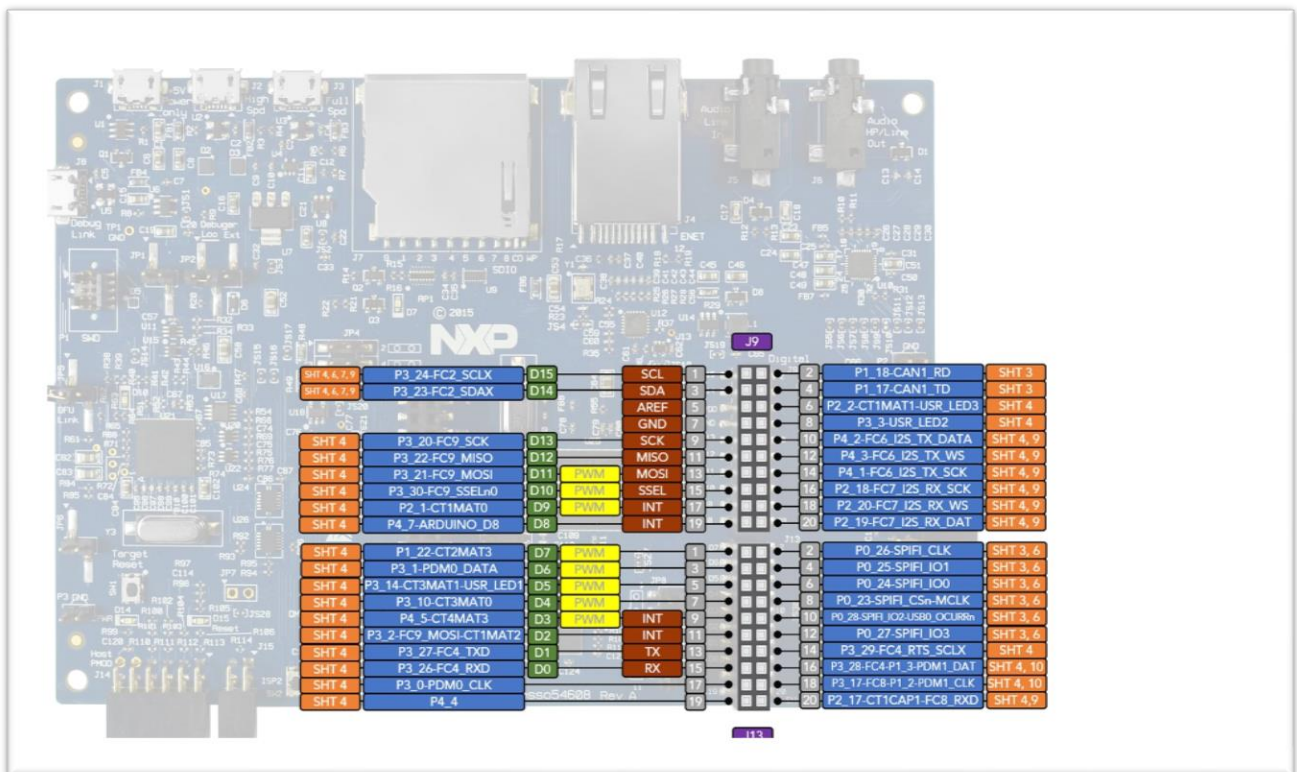


Figura 3. Pinout de la placa LPCXpresso54608 de NXP (1)

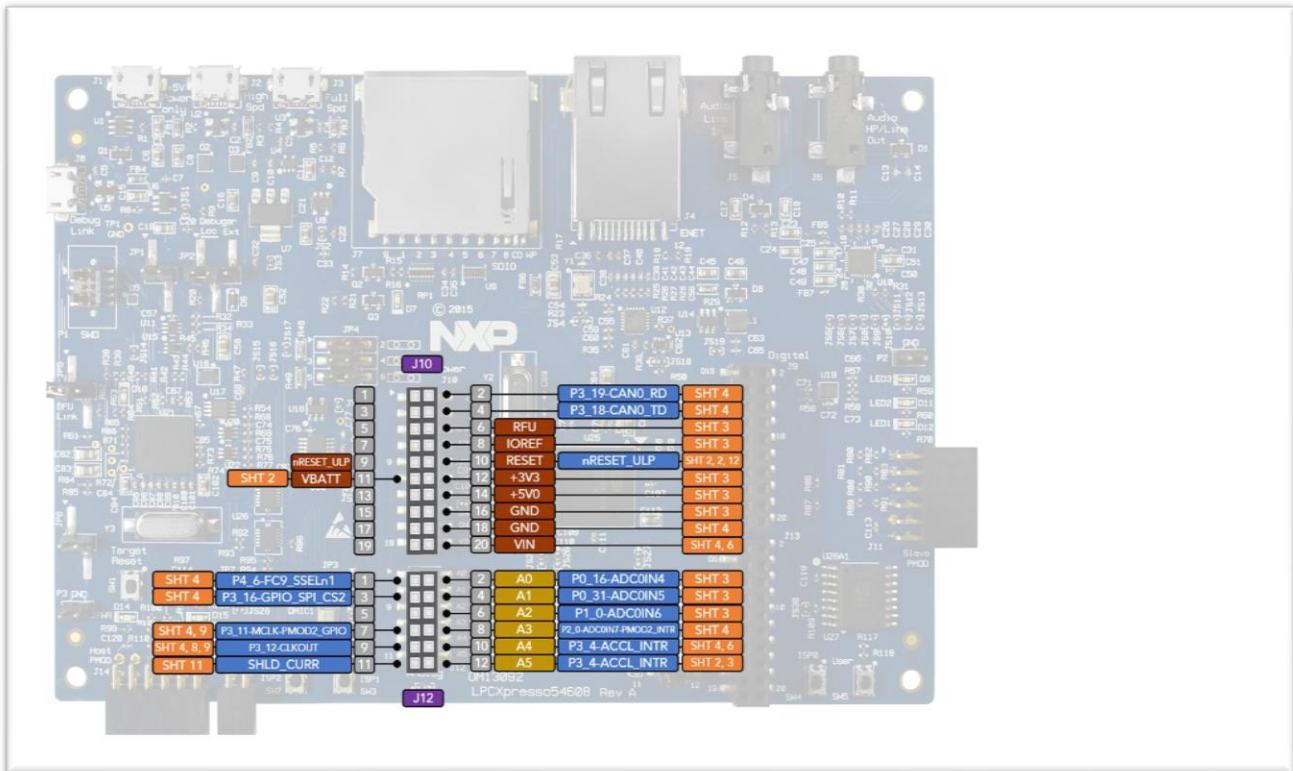


Figura 4. Pinout de la placa LPCXpresso54608 de NXP (2)

En estas imágenes tenemos toda la información necesaria acerca de los puertos, pulsadores, leds y pines de la placa.

En principio, para este proyecto usaremos mayoritariamente los puertos de ethernet y bus CAN, para la cual es necesaria una etapa de expansión en la placa, el acelerómetro propio de la placa para las pruebas, el puerto serie a través de USB y los leds.

### 2.1.5 Configuración de la placa para su uso con Mbed

Como se ha puntualizado antes la placa de desarrollo LPCXpresso54608 cuenta con soporte para Mbed que es el entorno donde vamos a trabajar y el RTOS que vamos a instalar en la placa.

Para poder trabajar con Mbed es necesario configurar la placa primero, para ello seguimos una sencilla guía que aparece en la página web de Mbed y que consiste en:

1. Conectar el puerto de Debug Probe de la placa al PC mediante un cable USB.
2. Descargar el driver para comunicación en serie más actualizado que proporciona Mbed.
3. Correr el instalador que se ha descargado asegurándose antes de que los jumpers de la placa están en la posición adecuada para el modo de configuración.
4. Una vez terminado el proceso de instalación, volver a colocar los jumpers de la placa en su posición estándar.

Y listo, el programa debería configurarlo todo correctamente y a partir de ahora nos debería aparecer la placa como un dispositivo de almacenamiento USB cada vez que lo conectemos al PC.

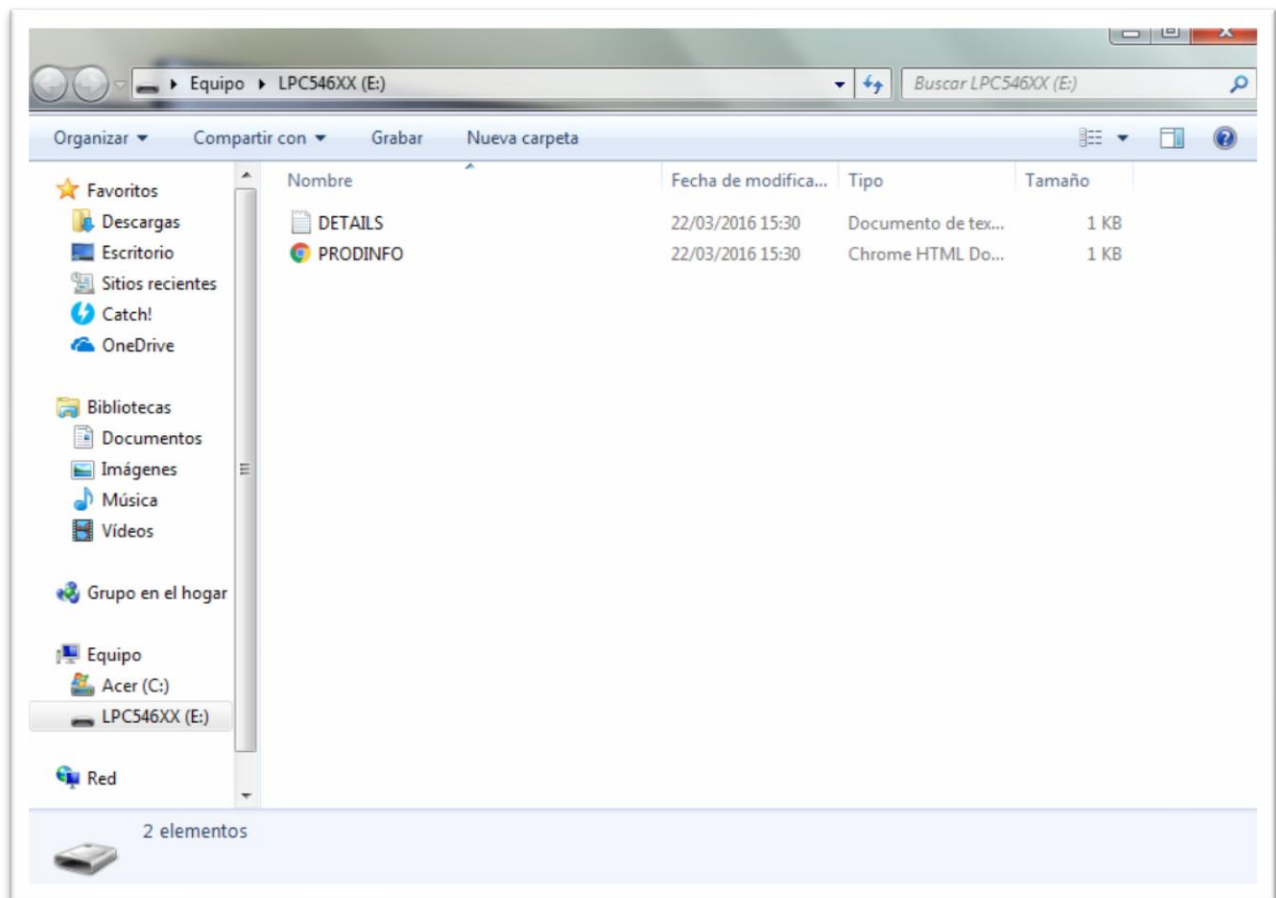


Figura 5. Captura de pantalla de la placa actuando como dispositivo de almacenamiento USB

Como se puede observar el proceso de configuración es muy sencillo y no requiere de grandes conocimientos lo cual es un aspecto positivo de usar Mbed como plataforma de desarrollo.

## 2.2. Etapa de expansión para la comunicación por bus CAN

Para poder implementar el uso del bus CAN en el proyecto es necesaria una etapa de expansión, esto es así debido a que los pines de CAN que tiene la placa de desarrollo LCPXpresso54608 son los pines CAN TX y CAN RX que no se corresponden con el bus que va a transmitir luego la información de un lugar a otro (CAN H y CAN L).

Para solucionar este problema se pensaba usar la expansión para bus CAN OM13099, pero debido a problemas con el entorno de desarrollo de Mbed, que se explican más adelante, no será necesario su uso sino el de la expansión CAN-BUS Shield V1.2 de DIGITAL.

Además, para el desarrollo de las pruebas y como alternativa a la placa LCPXpresso54608 de NXP se usará la placa de desarrollo NUCLEO-L4R5ZI-P, con esta placa se podrá observar la compatibilidad de Mbed y sus limitaciones.

## 2.2.1 CAN-BUS Shield V1.2 de DIGITAL

Esta expansión para bus CAN es muy útil ya que gracias al trabajo que realizan los dos microcontroladores con los que cuenta (MCP2551 y MCP2515) podemos usar un bus CAN mediante una comunicación SPI.

La disposición general de la expansión es la siguiente:

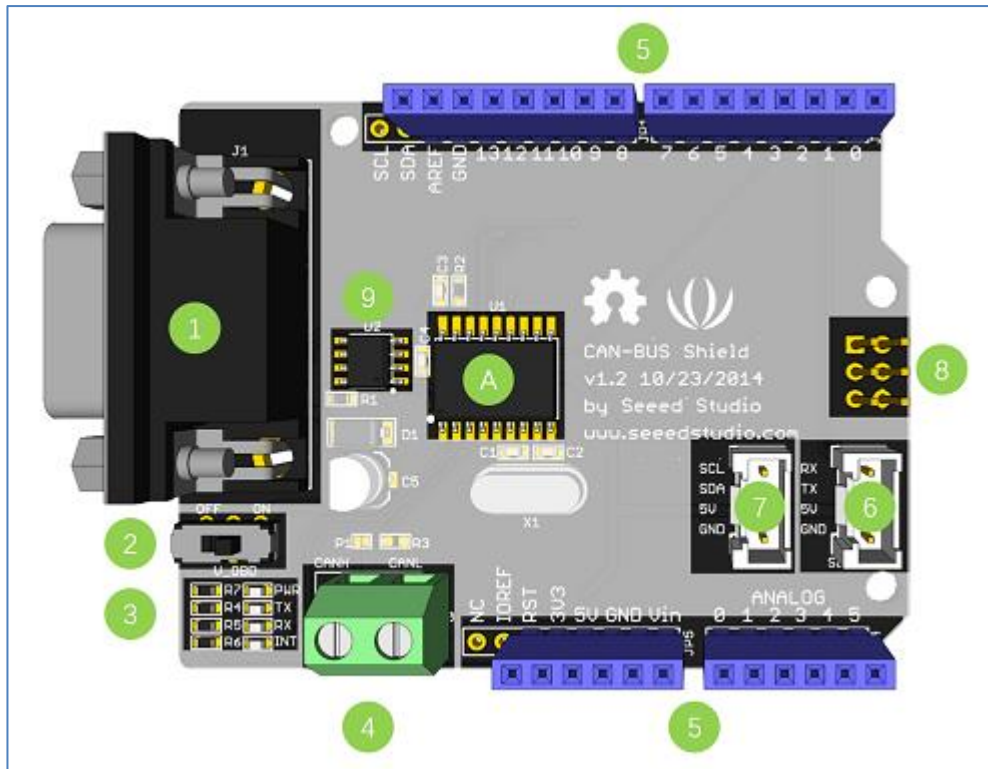


Figura 6. Partes de CAN-BUS Shield V1.2 de DIGITAL

1. Interfaz DB9: Puerto serie con conector del tipo DB9.
2. Interruptor para alimentación V\_OBD: Para obtener energía de la interfaz OBDII o no.
3. Indicadores LED:
  - a. PWR: Encendido cuando la placa esta alimentada.
  - b. TX: Parpadea cuando se están enviando datos.
  - c. RX: Parpadea cuando se están recibiendo datos.
  - d. INT: Interrupción de datos.
4. Terminales CANH y CANL.
5. Pin out.
6. Conector Serial Grove.
7. Conector I2C Grove.
8. Pines ICSP.
9. MCP2551: Transciever CAN de alta velocidad.
- A. MCP2515: Controlador CAN independiente con interfaz SPI.



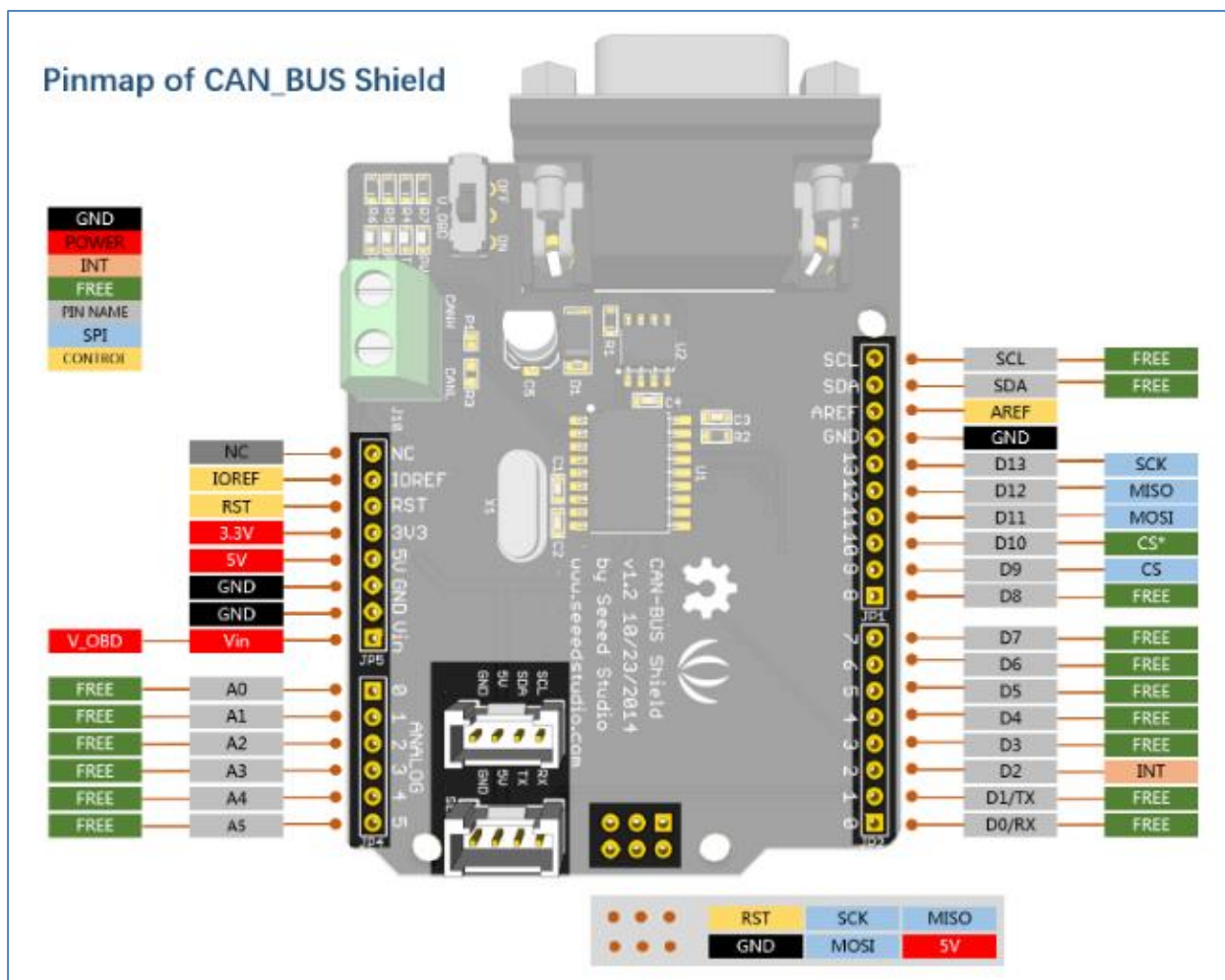


Figura 7. Pinout de CAN-BUS Shield V1.2 de DIGITAL

Viendo este pinout se pueden identificar rápidamente las partes importantes que son:

- Los pines de alimentación, tierra y referencia.
- Los pines de comunicación SPI:
  - MISO – D12
  - MOSI – D11
  - CS – D9
  - SCK – D13

Además de conocer las partes que componen esta expansión y el pinout de la misma también es importante saber un poco más en detalle cómo funcionan los microcontroladores MCP2515 y MCP2551 ya que en realidad son las partes que van a realizar la mayor parte del trabajo.

- MCP2515:

Este microcontrolador se encarga de controlar la transmisión de datos por el bus CAN a través de una comunicación SPI, es decir, a grandes rasgos realiza una transformación de SPI (MOSI, MISO, SCK, SS) a CAN (CANTX, CANRX).

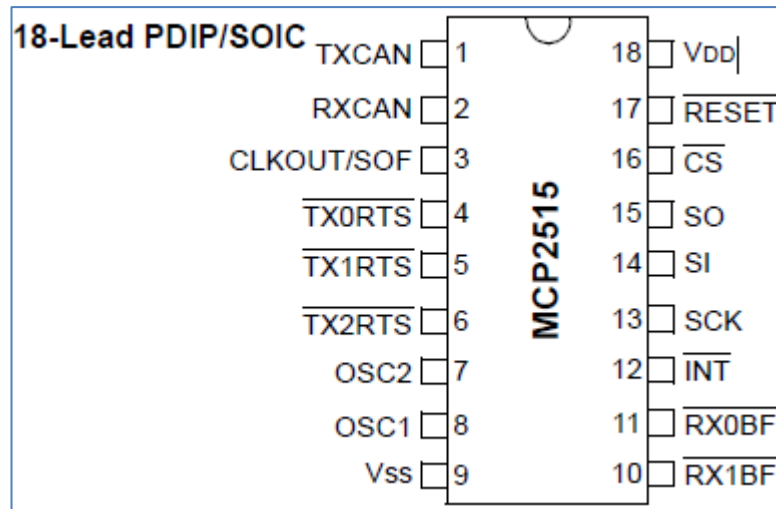


Figura 8. Pinout de MCP2515

Como se observa en el pinout el microcontrolador tiene las entradas de comunicación SPI y CAN (TX y RX) entre otras.

Pero aún no se tendrían los datos en el formato estándar para el envío por el bus CAN, para ello se necesitará un transceiver.

- MCP2551:

Este microcontrolador es un transceiver de alta velocidad que se encargará de transformar los datos de CAN (TX y RX) a CANH y CANL para poder leer y escribir en el bus CAN con este formato estándar.

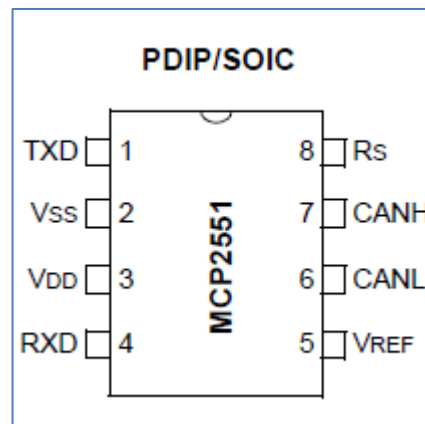


Figura 9. Pinout de MCP2551

Este micro también lo necesitaremos por separado para conectarlo directamente a la NUCLEO-L4R5ZI-P que cuenta con unos pines de CANRX y CANTX que necesitaremos transformar en CANH y CANL.

## 2.2.2 Placa de desarrollo NUCLEO-L4R5ZI-P

Esta placa de desarrollo se utilizará principalmente para aplicaciones con el bus CAN y para mostrar la compatibilidad de los códigos en Mbed, pero cuenta con muchas más utilidades que no se usarán en este proyecto.

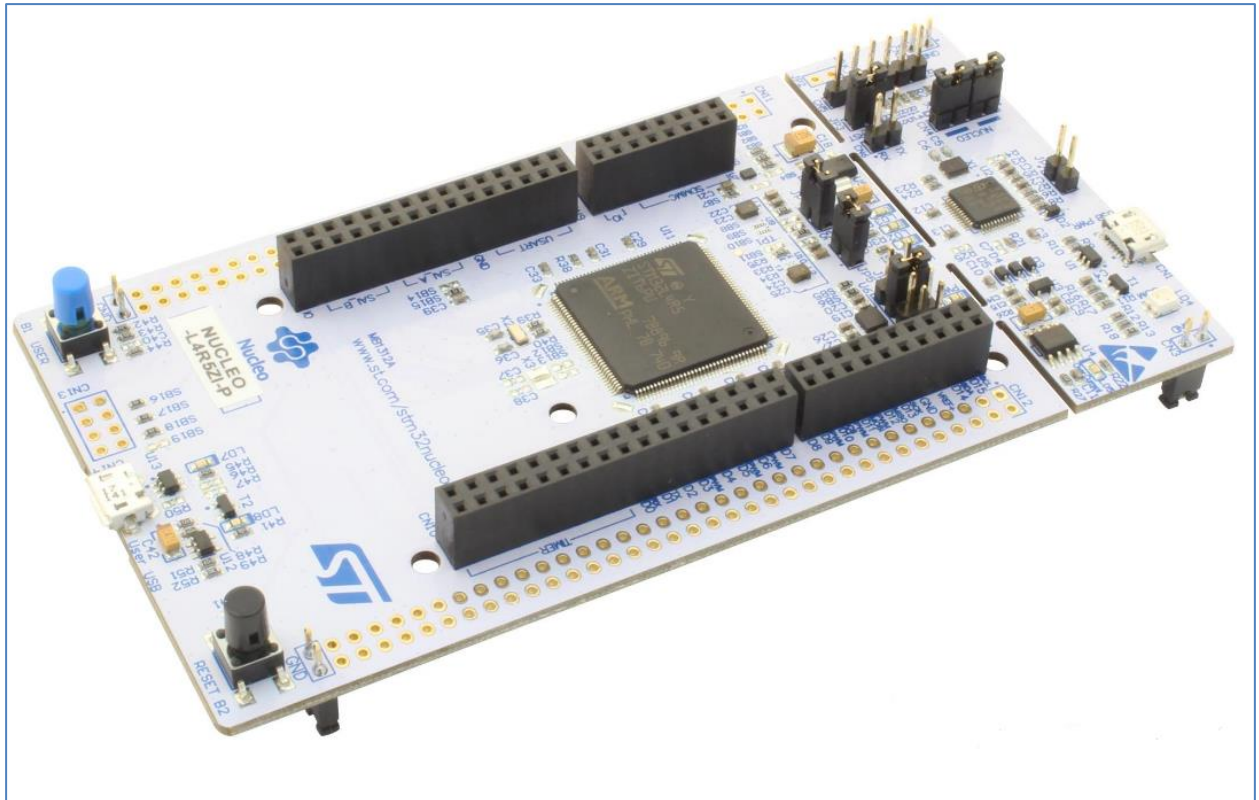


Figura 10. Placa de desarrollo NUCLEO L4R5ZI-P

- **Características del MCU**

El microcontrolador de la placa es de la familia STM32 y cuenta con las siguientes características:

- STM32L4R5ZIT6 en paquete LQFP144.
- CPU ARM®32-bit Cortex®-M4 con FPU.
- Frecuencia de CPU máxima de 120 MHz.
- VDD de 1,71 V a 3,6 V.
- Flash de 2MB.
- 640KB SRAM.
- RTC con calendario HW, alarmas y calibración.
- Hasta 24 canales de detección capacitivos: son compatibles con sensores táctiles, lineales y rotativos.
- Funciones gráficas avanzadas:
  - Chrom-ART Accelerator™ (DMA2D) para la creación mejorada de contenido gráfico.
  - Chrom-GRC™ (GFXMMU) que permite hasta un 20% de optimización de recursos gráficos.
  - Controlador MIPI® DSI Host con dos líneas DSI que funcionan a una velocidad de hasta 500 Mbits/s cada una.
  - Controlador LCD-TFT.
- Temporizadores 16x: 2 x 16 bits de control avanzado del motor, 2 x 32 bits y 5 x 16 bits para uso general, \* 2x 16 bits básicos, 2x temporizadores de baja potencia de 16 bits (disponibles en el modo de parada), 2x watchdogs, temporizador SysTick.
- Hasta 136 E/S rápidas, la mayoría de 5 V tolerantes, hasta 14 E/S con alimentación independiente hasta 1,08 V.
- Interfaz de memoria externa para memorias estáticas que admiten memorias SRAM, PSRAM, NOR,

NAND y FRAM.

- 2 x interfaz de memoria OctoSPI.
- 4x filtros digitales para modulador sigma delta.
- ADC de 12 bits a 5 Msps, hasta 16 bits con sobremuestreo de hardware, 200  $\mu$ A / Msps.
- DAC de 2x 12 bits, muestra de baja potencia y retención.
- Amplificadores operacionales 2x con PGA incorporado.
- 2x comparadores de potencia ultra baja.
- Interfaces de comunicación 20x.
- USB OTG 2.0 de velocidad completa, LPM y BCD.
- 2x SAIs (interfaz de audio en serie).
- 4x I2C FM + (1 Mbit / s), SMBus / PMBus.
- 6x USARTs (ISO 7816, LIN, IrDA, módem).
- 3x SPIs (5x SPIs con el OctoSPI dual).
- CAN (2.0B Active) y SDMMC.
- Controlador DMA de 14 canales.
- Generador de números aleatorios verdaderos.
- Unidad de cálculo CRC, ID única de 96 bits.
- Interfaz de cámara de 8 a 14 bits hasta 32 MHz (blanco y negro) o 10 MHz (color).

#### • Características de la placa

La placa cuenta con las siguientes funcionalidades y opciones:

- Dos tipos de recursos de extensión:
  - Conectividad con Arduino Uno Revision 3.
  - Cabezales de pines de extensión Morpho de STMicroelectronics para un acceso completo a todas las E/S de STM32.
- On-board ST-LINK / V2-1 depurador/programador con conector SWD.
  - Interruptor de modo de selección para usar el kit como un ST-LINK / V2-1 independiente.
- Opciones de alimentación flexibles: ST-LINK USB VBUS o fuentes externas.
- Tres LEDs de usuario.
- Dos pulsadores: USUARIO y RESET.
- Capacidad de re-enumeración de USB: tres interfaces diferentes compatibles con USB.
- Puerto COM virtual Almacenamiento masivo (unidad de disco USB) para la programación de arrastrar y soltar.
- Puerto de depuración

#### • Pinout de la placa

Aquí se presentarán las entradas y salidas de la placa y es importante saber que las etiquetas de estos pines que se pueden usar en Mbed son las que son de color celeste y azul marino.



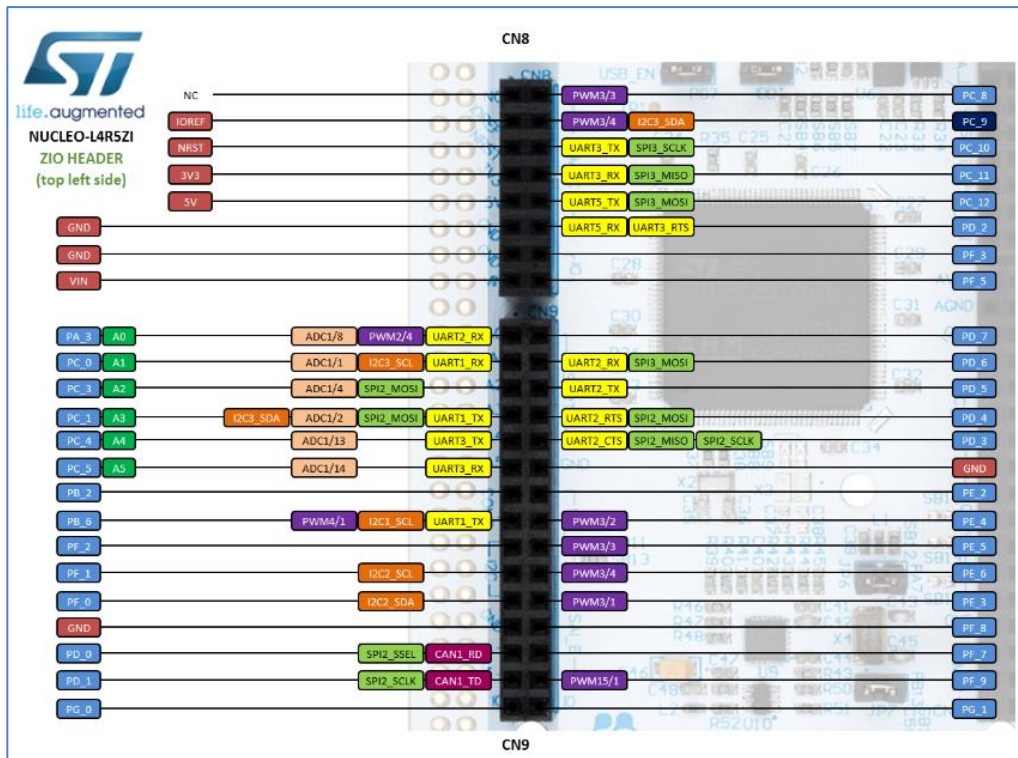


Figura 11. Pinout de las hileras CN8 y CN9

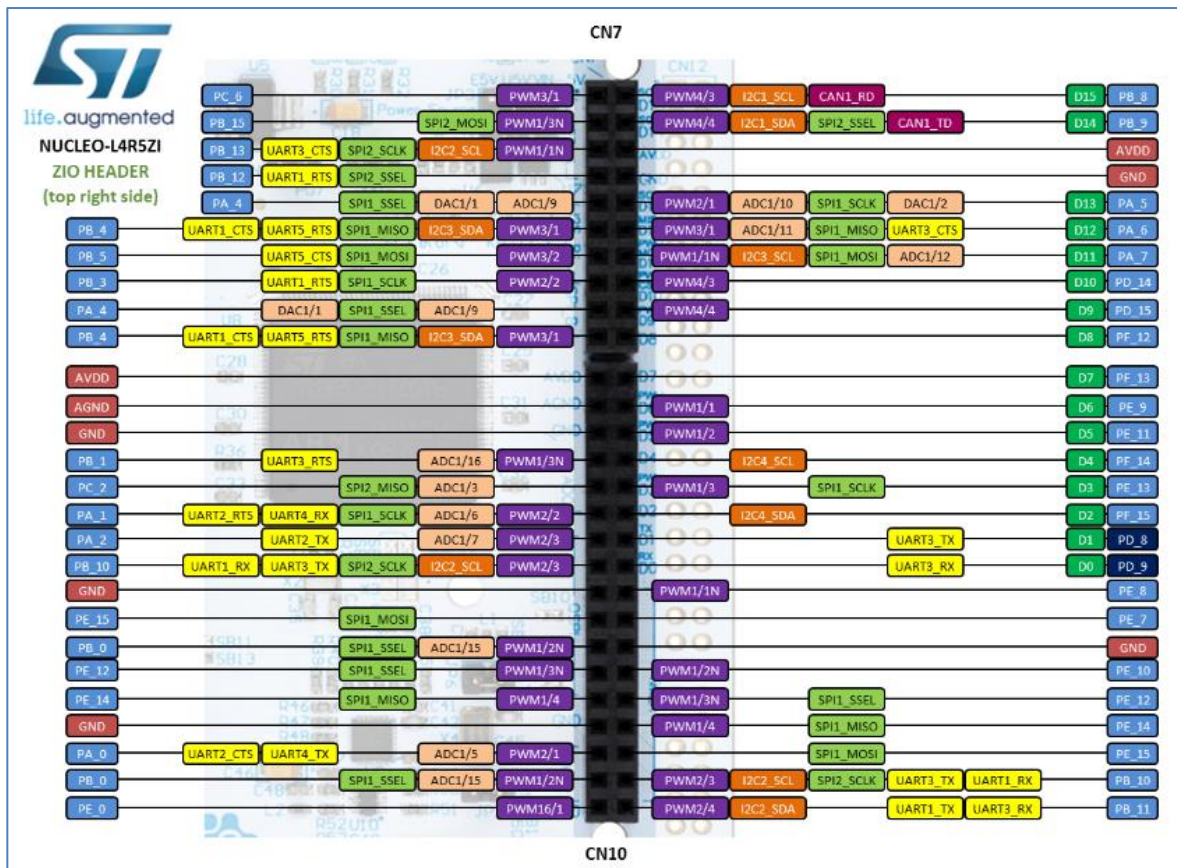


Figura 12. Pinout de las hileras CN7 y CN10

Además de estos pines existen dos hileras más que vienen sin conectores hembras, pero podrían utilizarse también, cosa que no se hará ya que no se necesita para este proyecto.

- **Configuración de la placa**

Esta placa ya viene configurada para su uso en Mbed por lo que no es necesario que se realice ningún proceso para poder utilizarla en esta plataforma, pero si fuera necesario configurarla por algún motivo solo tendríamos que seguir un proceso muy similar al que se explicó para la placa LPCXpresso54608.

# 3 MBED, EL RTOS DE CÓDIGO LIBRE PARA EL DESARROLLO DE APLICACIONES IOT

---

**M**bed es una plataforma de desarrollo y RTOS para dispositivos de baja potencia conectados a Internet basados en microcontroladores ARM Cortex-M. Estos dispositivos se conocen como dispositivos IoT y, como se explica en la introducción, cada vez están tomando más importancia en el mundo.

## 3.1 Descripción general

La elección de Mbed como plataforma de desarrollo tiene un motivo fundamental y es que Mbed permite que el programador trabaje el lenguaje de alto nivel C/C++ y se abstraiga del microcontrolador que está usando, esto hace más sencillo programar y, además, hace que siempre se programe igual un microcontrolador, sin importar el fabricante, por lo que es mucho más universal y, en teoría, permite que las aplicaciones desarrolladas para una plataforma se puedan utilizar sin problemas en otra, siempre que tenga soporte para Mbed.

Otra de las cosas interesantes de Mbed es que las aplicaciones que se desarrollan para ella se pueden hacer usando el IDE en línea que la propia Mbed provee, esto nos da la ventaja de no tener que instalar el IDE en nuestro PC, sino que solo teniendo un navegador web podremos programar y compilar código en línea y el propio navegador se encargará de descargar el archivo binario correspondiente a nuestra aplicación.

El hecho de que Mbed sea de código libre y gratuito es también una característica a tener en cuenta ya que cualquiera puede acceder a ella y ahorrará costes.

### 3.1.1 Arquitectura y funcionalidades

Como se ha dicho antes, Mbed proporciona una capa que interpreta el código de la aplicación de manera que el hardware pueda “entenderlo” y esto lo consigue mediante las APIs del sistema operativo Mbed.

Sabiendo esto podemos entender mejor la arquitectura básica de una placa Mbed:

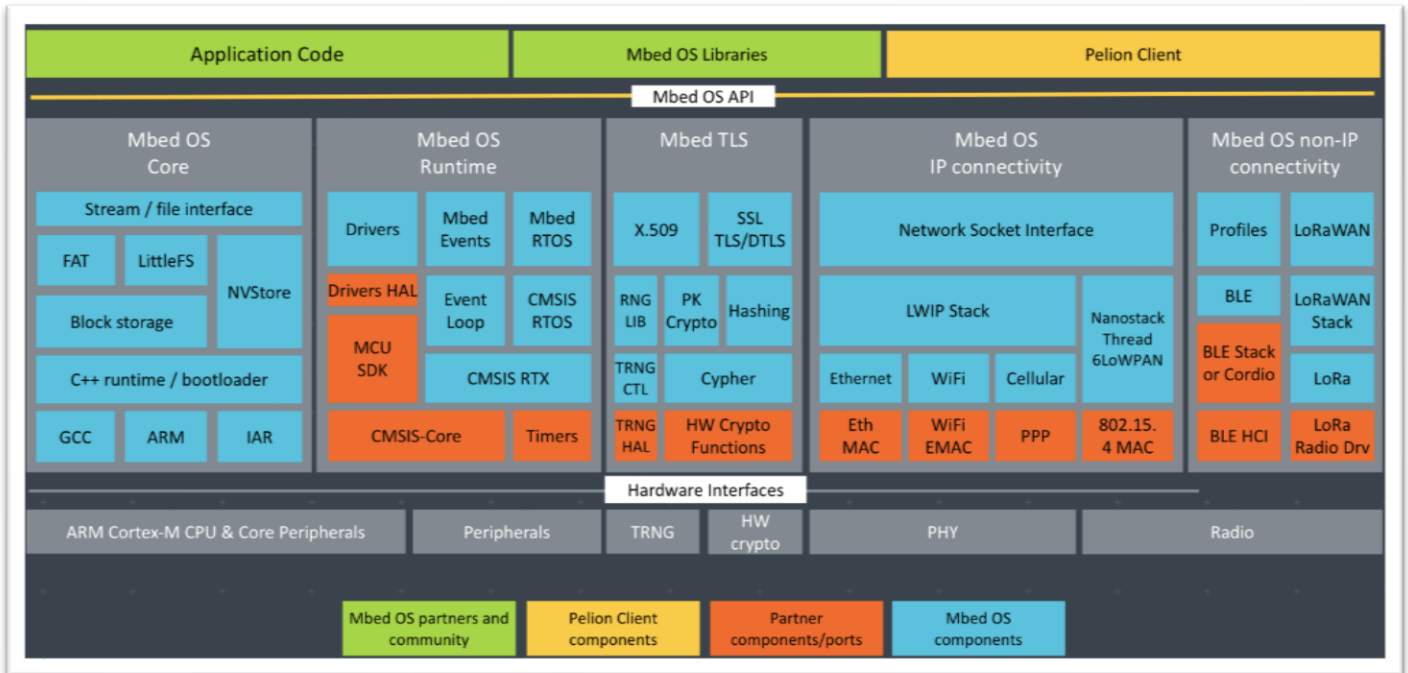


Figura 13. Arquitectura básica de una placa Mbed

Como vemos tiene distintas secciones y APIs, que detallaremos en los siguientes apartados.

### 3.1.1.1 Conectividad

Mbed ofrece un núcleo estable de tecnologías de conectividad establecidas y además agrega nuevas versiones en lanzamientos de funciones que hace trimestralmente de manera que los usuarios de la plataforma estén informados de las tendencias de la industria.

Las pilas de redes, conectividad y subprocesos de Mbed están certificadas debido a su madurez e interoperabilidad, y pueden satisfacer los diseños de IoT más exigentes debido a su flexibilidad ya que cubren una gran variedad de usos.

Mbed OS soporta la plataforma Pelion IoT, la cual nos permite administrar los dispositivos que tengamos desplegados y los datos.

### 3.1.1.2 Seguridad

La plataforma Pelion IoT tiene seguridad incorporada en todos los niveles, pero especialmente en la protección contra violaciones y la mitigación de sus consecuencias. Mbed ofrece servicios en la nube reforzados, pilas de comunicación robustas, actualizaciones de firmware robustas y dos bloques embebidos de seguridad específica: ARM Mbed TLS y Secure Partition Manager (SPM) que cumple con las mejores prácticas de la industria como parte de la arquitectura de seguridad de la plataforma de Arm. Mbed TLS protege los canales de comunicación entre un dispositivo y la puerta de enlace o el servidor, y usa un administrador de particiones seguro y dominios de seguridad aislados para reducir el área de ataque. En conjunto, esto proporciona un modelo único de seguridad de chip a nube.

### 3.1.1.3 Actualización remota de firmware

Mbed ofrece una integración completa de los servicios de actualización en línea de Pelion por lo que es capaz de actualizar una aplicación o la versión del sistema operativo de cualquier dispositivo conectado. La aplicación CLI de ARM Mbed da la posibilidad de crear cargas útiles de actualización, generar sus manifiestos y mandarlos a los dispositivos usando solo 2 comandos.

### 3.1.1.4 Hardware

El hardware que es compatible con Mbed está limitado ya que no todos los dispositivos lo soportan aun, pero tanto los desarrolladores como los usuarios de Mbed ayudan a que cada vez haya más dispositivos compatibles.

El hardware que se puede encontrar en Mbed es principalmente de 3 tipos:

- **Módulos:** Son los sistemas más grandes e incluyen un microcontrolador, conectividad centrada IoT y memoria en placa, son perfectos para diseñar productos IoT.
- **Placas:** Son una forma económica de empezar a desarrollar en Mbed y es la opción que se ha elegido para el proyecto debido a su versatilidad y precio (LPCXpresso54608).
- **Componentes:** Son el hardware más específico que vamos a añadir a nuestro proyecto y pueden tener como objetivo comunicaciones, medidas, actuación, etc. La base de datos de componentes de Mbed aloja librerías reutilizables para todo tipo de componentes.

### 3.1.1.5 Herramientas

Mbed tiene a disposición del usuario un paquete de herramientas bastante interesante que incluye:

- El compilador en línea que nos permite programar y compilar sin necesidad de descargar nada más en nuestro PC.
- El Arm Mbed CLI que nos permitirá trabajar en Python con líneas de comandos y sin necesidad de acceso a internet.
- Las herramientas de depuración, DAPLink y pyOCD que permiten programar y depurar muchos dispositivos.
- Las herramientas de validación del sistema operativo Mbed, Greentea y utest, para validar el proyecto una vez está terminado.

### 3.1.1.6 Documentación

Mbed cuenta con tres tipos de documentos que son referencias, tutoriales y guías de portabilidad. Las referencias son de mucha ayuda ya que se tratan de material técnico sobre las APIs y la arquitectura de Mbed, cosa que es muy útil sobre todo en el caso de las APIs. Los tutoriales muestran paso a paso como realizar tareas específicas, como por ejemplo configurar la comunicación serie de una placa de desarrollo. Y las guías de portabilidad muestran como trasladar Mbed OS a otros dispositivos.

## 3.1.2 Interfaces de Programación de Aplicaciones o APIs

Como ya sabemos las APIs son un conjunto de rutinas que proveen acceso a funciones de un determinado software, en nuestro caso Mbed, y cada una de ellas viene acompañada con un apartado de instrucciones donde se nos explica cómo funciona dicha API en la plataforma.

Existen distintos tipos de APIs en Mbed:

### 3.1.2.1 APIs de plataforma

Estas APIs son las más básicas de todas ya que proporcionan una infraestructura de administración del microcontrolador de propósito general, dando una experiencia más consistente al usuario ya que este contará con estructuras de datos y herramientas comunes que seguro necesitará.

Existen muchas APIs de este tipo, pero como ejemplo de ellas vamos a elegir algunas que son especialmente útiles y se utilizarán en el proyecto:

- **Wait:** Una API que proporciona capacidades de espera simples. Al utilizar esta API nuestra aplicación esperara el tiempo declarado en ella en segundos y esto puede permitir el ahorro de energía ya que podríamos poner el microcontrolador en suspensión si todos los subprocesos están inactivos.
- **Callback:** Un Callback es una función proporcionada por el usuario que un usuario puede pasar a una

API, es decir, el Callback permite que la API ejecute el código del usuario cuando lo necesitemos. Básicamente esta API permite que podamos llamar una parte de nuestro código cuando lo necesitemos desde otra parte de nuestro código, consigue este objetivo administrando los punteros que ya conocemos de C/C++ para que no tengamos que hacerlo nosotros.

### 3.1.2.2 APIs de drivers

Estas APIs incluyen las entradas y salidas de la placa de desarrollo y las interfaces digitales, esto hace que podamos trabajar con ellas de manera muy sencilla para comunicarnos a través de las entradas y salidas con elementos del exterior como un PC u otros dispositivos que conectemos a la placa, con esto podremos medir o controlar desde la tensión en los pines de entrada y salida hasta un bus de comunicación CAN, pasando por el duty cycle de una salida PWM.

La cantidad de APIs de este tipo también es muy alta por ello se han escogido algunas de las que se usarán en el proyecto como ejemplo:

- **DigitalOut:** Con esta interfaz se puede controlar una salida digital de la placa de desarrollo que estamos usando, poniendo su valor a 0 o a 1, un ejemplo sencillo de una de estas salidas es un led de usuario que tenga la placa.
- **Timer:** Esta interfaz permite crear, iniciar, leer y detener un temporizador cuando queramos para medir tiempos con una precisión mayor de milisegundos, una de las utilidades más interesantes de esta API es que podemos interactuar con cualquier temporizador que hayamos creado independientemente, es decir, sin afectar al resto que seguirán funcionando normalmente.
- **Ticker:** Esta interfaz nos permite configurar una interrupción que ocurra de manera recurrente, es decir, podemos llamar una función repetidamente a una velocidad determinada. Al igual que la API Timer, esta interfaz también nos permite trabajar con cada ticker de manera independiente y tener por tanto múltiples interrupciones repetitivas con esta misma interfaz.
- **Serial:** Esta interfaz permite configurar de manera muy sencilla la funcionalidad UART y por lo tanto poder enviar y recibir datos por un puerto serie asíncrono. Una de las conexiones en serie que proporciona es la conexión con el puerto USB, lo que nos permite comunicarnos fácilmente con nuestro PC y poder trabajar más cómodamente. Entre muchas otras funcionalidades que nos aporta esta API tenemos la de configurar la velocidad de transferencia, ver si hay espacio disponible en el buffer de escritura, escribir o leer.
- **SPI:** Esta interfaz nos proporciona un maestro para el estándar Serial Peripheral Interface (SPI), de manera que la podemos utilizar para comunicarnos con esclavos que usen este estándar con las memorias flash, las memorias RAM o las pantallas LCD. Este estándar de comunicación nos permite tener una comunicación serie síncrona y que con un maestro definido por esta API podamos tratar con más de un esclavo. Dentro de las muchísimas posibilidades de esta API tenemos la de configurar la velocidad de transferencia y el formato de transferencia en bytes, por ejemplo.
- **CAN:** Esta interfaz nos permite trabajar con el estándar Controller-Area Network (CAN), este estándar permite comunicar microcontroladores y dispositivos entre si sin tener que pasar por un host. Esta interfaz nos permite escribir en el puerto CAN y recibir de él de manera muy sencilla y también configurar la frecuencia de reloj que estemos usando para ello.

### 3.1.2.3 APIs de RTOS

Estas APIs de RTOS nos proporcionan funcionalidades como la gestión de objetos como subprocesos, la sincronización de objetos y temporizadores, lectura de los ticks del sistema operativo o el reporte de errores del RTOS.

Algunas de las APIs de RTOS son: **Thread**, que define, crea y controla subprocesos paralelos, **Mutex**, que permite sincronizar la ejecución de hilos o subprocesos, o **Semaphore**, que administra el acceso de subprocesos a un conjunto de recursos compartidos de un determinado tipo.

### 3.1.2.4 APIs de USB

Son las APIs que proporcionan una comunicación por medio de un USB con cualquier otro dispositivo, ya sea para una comunicación serie básica o para controlar un teclado o un ratón, por ejemplo. Se podrían considerar APIs de Drivers ya que al fin y al cabo realizan la misma función solo que usando el USB como medio de comunicación, son específicas ya que nos permiten una comunicación sencilla entre nuestro PC y nuestra placa de desarrollo.

Dentro de la APIs de esta categoría podemos encontrar:

- **USBSerial:** Esta API se puede utilizar para emular un puerto serie a través del USB.
- **USBAudio:** Esta interfaz nos permite enviar y recibir datos de audio a través de una conexión USB, seleccionando Mbed Audio como micrófono en nuestro PC podremos obtener datos de audio desde la placa de desarrollo y seleccionando en nuestro PC, Mbed Audio como altavoz podremos enviar datos de audio desde el PC a la placa de desarrollo.
- **USBKeyboard:** Esta API nos provee de la funcionalidad de un teclado a través del USB.

### 3.1.2.5 APIs de sockets de red

La API de socket de red es la interfaz de programación de aplicaciones para redes IP, esta API se relaciona con la capa de transporte en internet y en el caso de Mbed es abstracta por lo que puede usar protocolos como TCP, UDP y entrega de datos no IP para redes NB-IoT.

Para poder usar estas APIs se deben seguir los siguientes pasos:

1. Inicializar una interfaz de red, como por ejemplo ethernet.
2. Crear un socket, por ejemplo, usando el protocolo TCP.
3. Realizar la conexión.
4. Enviar datos.
5. Recibir datos.
6. Cerrar el socket.

Dos de las APIs de Mbed más importantes en esta categoría son:

- **TCPsocket:** Que proporciona la capacidad de enviar y recibir un flujo de datos a través del protocolo TCP.
- **UDPSocket:** La cual proporciona la capacidad de enviar y recibir paquetes de datos a través del protocolo UDP.

### 3.1.2.6 APIs de interfaces de red

Son las APIs necesarias para poder abrir un socket de red con las interfaces explicadas en el punto anterior, ya que necesitamos determinar en qué interfaz de red vamos a crear el socket. Con estas APIs el usuario puede elegir controlador, método de conectividad y pila IP.

Algunos de las interfaces de red disponibles en Mbed son:

- **Ethernet:** Por defecto, esta API en C++ para la conexión a internet a través de Ethernet no requiere ninguna configuración ya que, es capaz de seleccionar el controlador Ethernet predeterminado para el destino y seleccionar la pila de red correcta.
- **Wi-Fi:** Esta API nos permite conectarnos a internet a través de un dispositivo Wi-Fi de manera muy sencilla.

### 3.1.2.7 APIs de almacenamiento

Estas APIs contienen las interfaces para las operaciones con el sistema de archivos y los dispositivos de bloque que proporcionan el almacenamiento bruto para los sistemas de archivo.

Las APIs de almacenamiento disponibles en el sistema operativo de Mbed son:

- **KVStore**: una interfaz común para los componentes que presentan set / get API.
- **File System**: una interfaz común para el uso de sistemas de archivos en dispositivos de bloque.
- **Block device**: una interfaz común para dispositivos de almacenamiento basados en bloque.
  - Dentro de este punto nos va a interesar especialmente la API llamada: **SDBlockDevice** que nos proporciona un controlador de dispositivo de bloque para tarjetas SD y chips de memoria eMMC. Las tarjetas SD o los chips eMMC ofrecen una capa FTL completa sobre la memoria flash NAND. Esto hace que el almacenamiento sea adecuado para sistemas que requieren sobre 1 GB de memoria. Además, las tarjetas SD son una forma popular de almacenamiento portátil y son útiles si se desea almacenar datos a los que poder acceder desde un PC.
- **PSA protected storage**: API compatible con PSA para un entorno de procesamiento no seguro (NSPE).
- **PSA internal storage**: API compatible con PSA para un entorno de procesamiento seguro (SPE).

### 3.1.2.8 APIs de Bluetooth, LoRaWAN, NFC y seguridad

Son APIs que nos permiten trabajar con las funcionalidades que sus propios nombres indican y tienen aplicaciones muy interesantes, en especial las relacionadas con la seguridad, pero al no tener relevancia en este proyecto no se va a profundizar en ellas.

## 3.2 Entorno de Desarrollo Integrado o IDE

La plataforma de Mbed cuenta con dos opciones a la hora de empezar a trabajar con ella:

- **Arm Mbed Studio**: el IDE de escritorio de Mbed que está disponible para cualquier persona que tenga una cuenta en Mbed.
- **Arm Mbed Online Compiler**: es el compilador en línea de Mbed y es que se ha elegido para el proyecto ya que no es necesario descargar nada y así no dependemos tanto de la capacidad de procesamiento de nuestro PC, además de que podemos disponer de los programas desarrollados en cualquier lugar con conexión a internet.

### 3.2.1 Vistazo a Arm Mbed Online Compiler

En la siguiente figura se puede apreciar el aspecto que presenta el entorno de programación online de Mbed.

El entorno consta de una barra de herramientas principal, una ventana con los programas o archivos que tenemos en nuestro perfil online, una ventana de información con las últimas modificaciones y la ventana principal de programación.



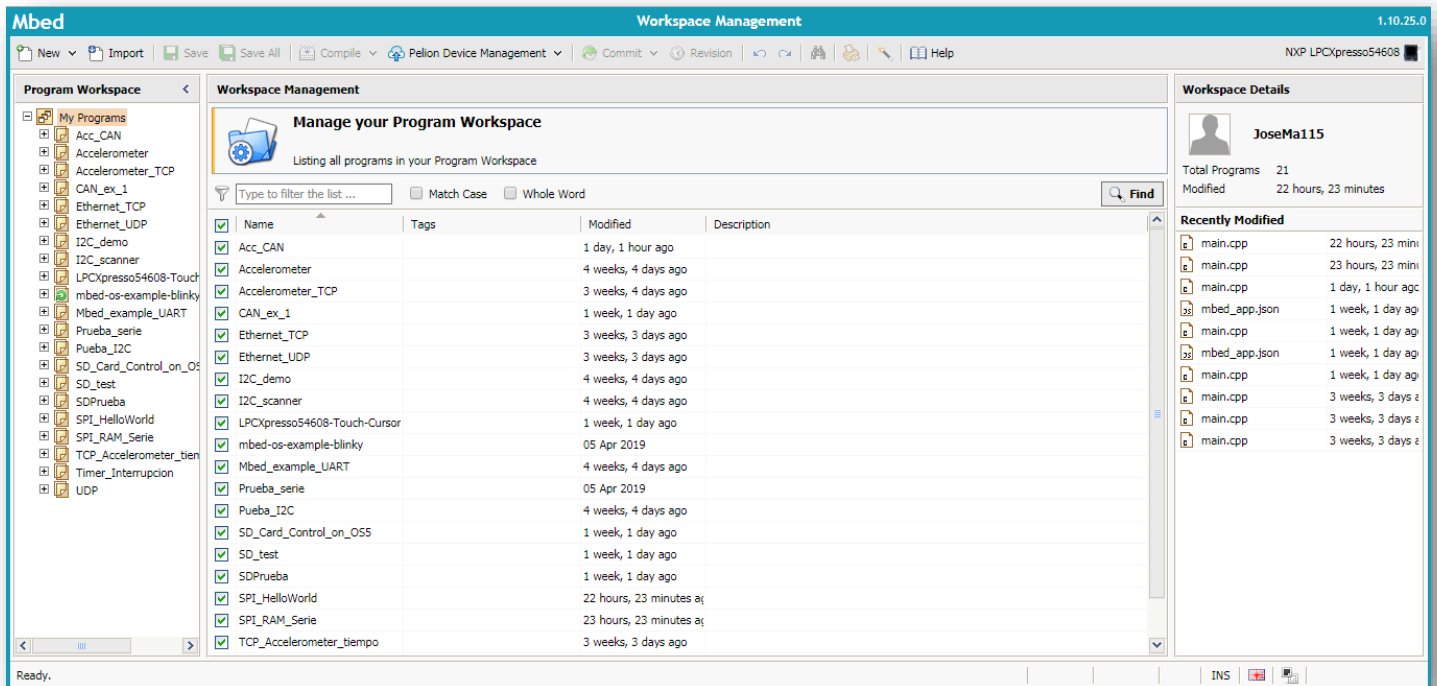


Figura 14. Entorno de programación Arm Mbed Online Compiler

Vamos a empezar fijándonos en la barra de herramientas donde podemos observar las diferentes utilidades que tenemos en este IDE.

Primero vemos la opción de *New* con la cual podemos crear nuevos programas, archivos o librerías.

A continuación, vemos el botón de *Import* mediante el cual podemos importar archivos, programas y librerías a nuestro espacio de trabajo desde la nube.

Luego tenemos *Save* y *Save as* cuya función es la de realizar un guardado de todas las modificaciones realizadas en el programa.

La siguiente es *Compile*, que es la herramienta que creará, a partir de nuestro código en C++, el archivo binario con el que podremos programar nuestra placa de desarrollo.

*Peblion Device Management* nos da la posibilidad de trabajar con el gestor de dispositivos del mismo nombre.

*Commit* nos permite hacer un commit a la web de Mbed y *Revision* nos permite realizar una revisión del programa seleccionado.



Figura 15. Barra de herramientas (Parte 1)

*Retroceder* y *Avanzar* funcionan, como es de esperar, dando la posibilidad de deshacer algo que hemos tocado en el código y recuperar lo desechado.

*Find* nos permite encontrar palabras en el código.

*Print* para obtener un archivo de texto de nuestro código.

*Format Code*, que es el icono de la varita mágica, nos permite poner el código con las tabulaciones y el montaje más correcto para poder entenderlo.

Y *Help* que nos dará la información que necesitaremos sobre cualquier parte del IDE.

Por último, el campo en el que aparece el nombre de la placa que estamos usando nos permite cambiar de placa cuando queramos y acceder a la información de esta.



Figura 16. Barra de herramientas (Parte 2)

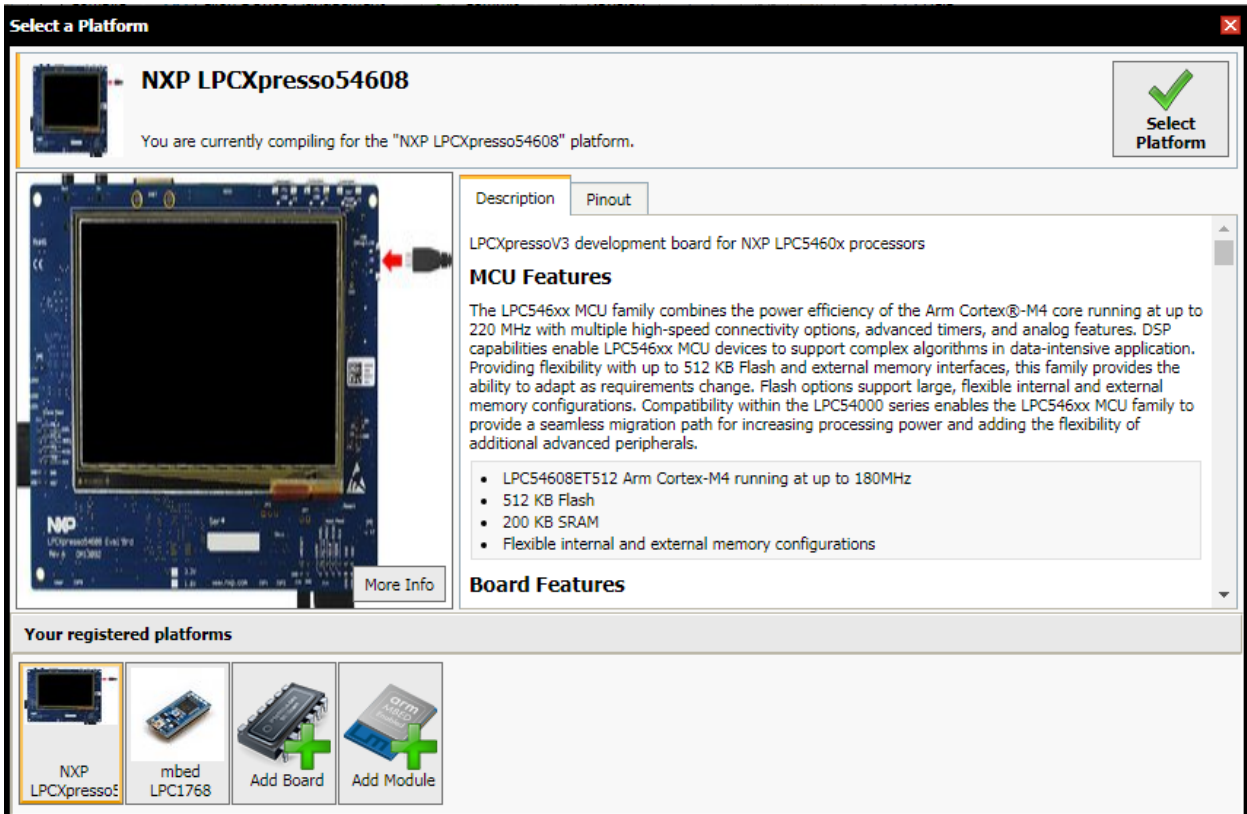


Figura 17. Ventana de selección de plataforma

Ahora fijándonos en la parte izquierda del IDE vemos el *Program Workspace* una ventana en la que nos aparecen los programas que hemos realizado y clicando en cada uno podemos ver las partes que las componen como por ejemplo las librerías propias del sistema operativo, librerías externas añadidas, programas principales o archivos de cabecera.

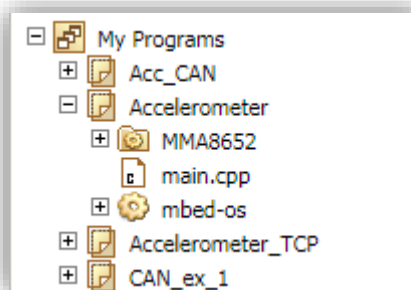


Figura 18. Program Workspace

Y ya solo nos queda por destacar la ventana principal del IDE donde aparece el archivo que tengamos

seleccionado en el workspace y su código, además es donde podemos introducir nuevas líneas de código y borrar o modificar las ya existentes, es donde vamos a trabajar la mayor parte del tiempo.

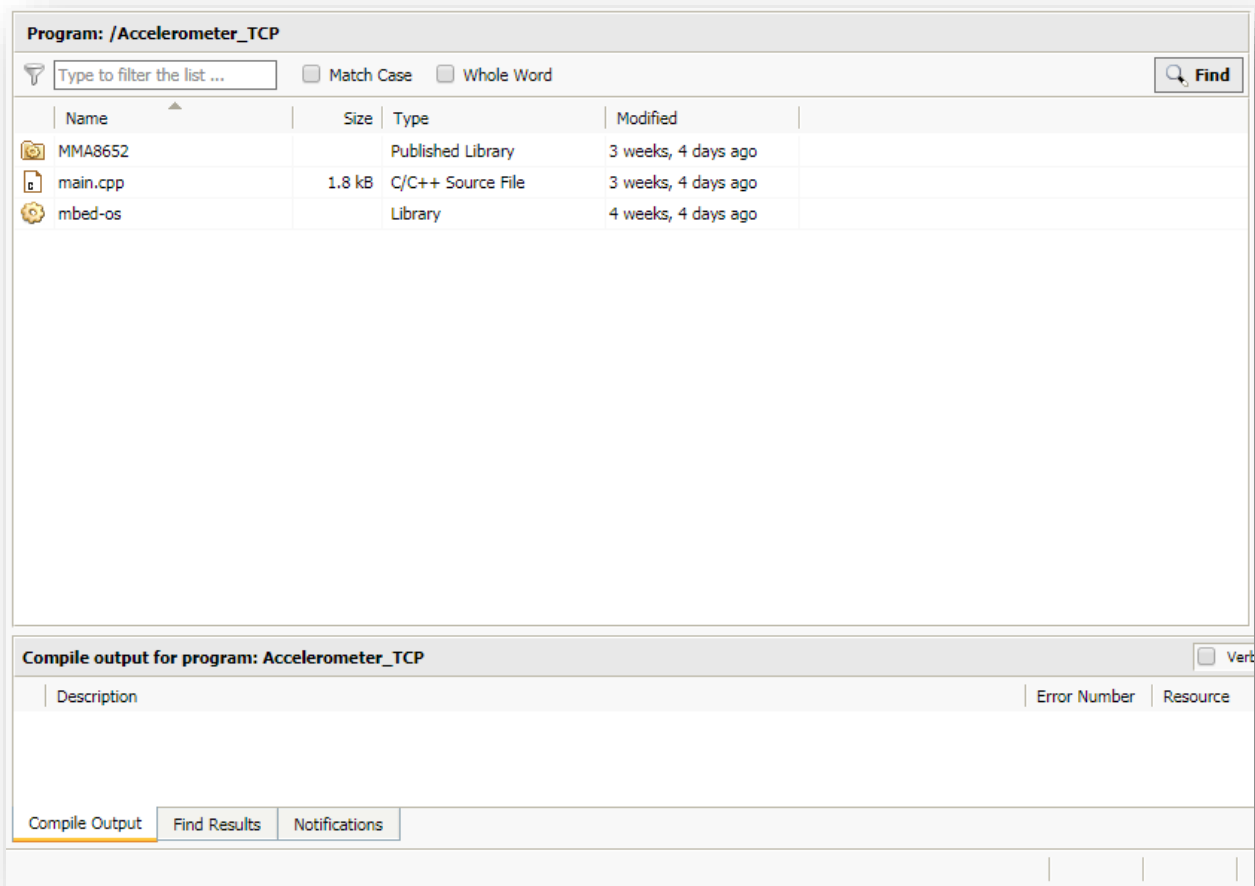


Figura 19. Ventana principal del IDE mostrando los archivos internos de un programa

Como se puede observar este IDE no es muy diferente de otros que pertenezcan a una empresa que fabrique las placas de desarrollo, la única diferencia notable va a ser la de programar a más a alto nivel ya que contamos con las librerías que nos proporciona el propio sistema operativo, donde podemos encontrar las clases de datos que se corresponden con las APIs que se explicaron en el apartado anterior, cosa que nos facilitará bastante el trabajo en muchos casos.

Además, se puede ver que es muy sencillo adaptar un código ya creado para otra placa que soporte Mbed como RTOS, lo único que tenemos que hacer es cambiar la plataforma elegida en la barra de herramientas, mirar su pinout y modificar las entradas y salidas que sean necesarias.

### 3.2.2 Ejemplo práctico de uso

En este apartado se va a explicar el funcionamiento básico de la plataforma de Mbed mediante un ejemplo sencillo, en el que se va a programar un parpadeo de leds en la placa.

Para ello deberemos tener la placa configurada como se explicó en apartados anteriores de forma que al conectarla al PC mediante un USB nos aparecerá como un dispositivo de almacenamiento ampliado.

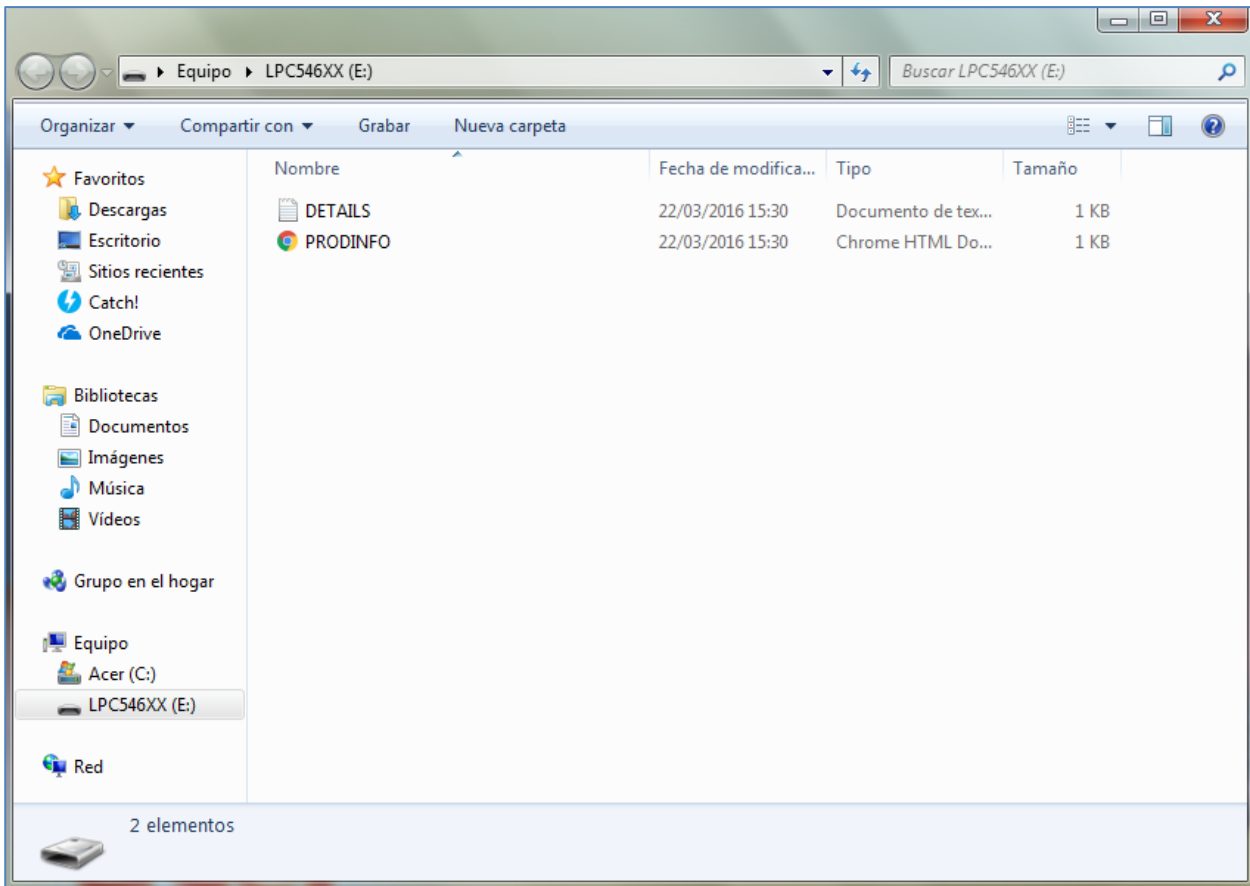


Figura 20. El PC detecta la placa de desarrollo como almacenamiento ampliado

A continuación, abrimos el entorno de programación online de Mbed y creamos un nuevo programa en el que solo necesitaremos incluir dos cosas: la librería del sistema operativo Mbed, que nos dará acceso a clases de datos que necesitaremos para todos los programas que realicemos, y un archivo main.cpp que será donde se encuentre el código principal de nuestro programa.

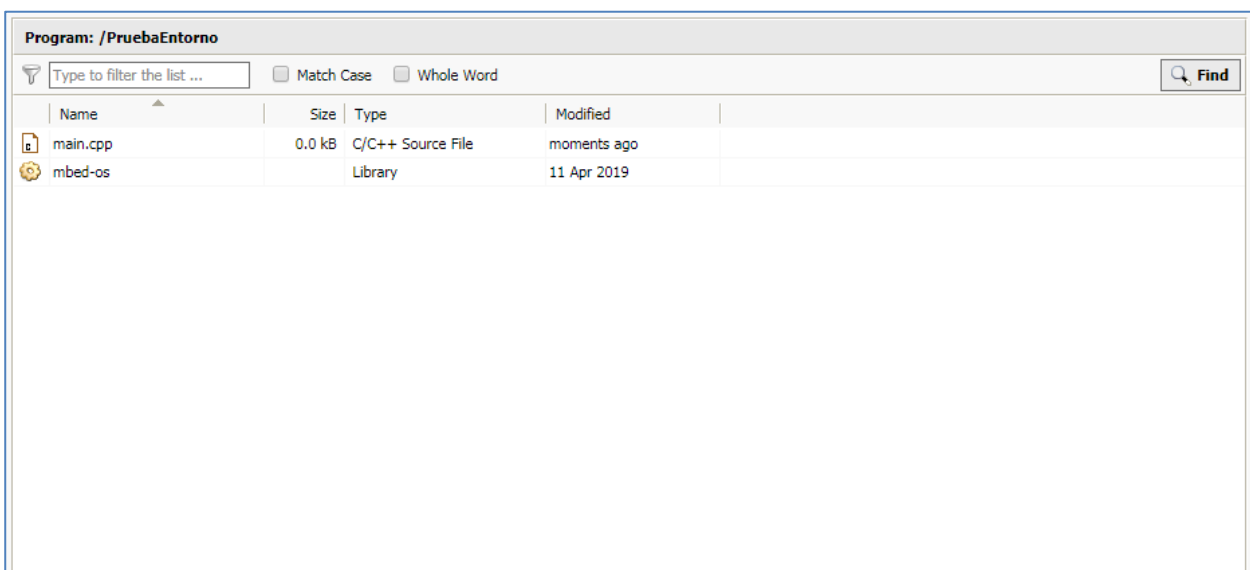


Figura 21. Programa PruebaEntorno en Arm Mbed Online Compiler

En el archivo main.cpp escribimos nuestro código que en este caso deberá conseguir que el led 1 de la placa

cambie de estado cada medio segundo y que el led 2 lo haga cada dos segundos.

Para ello simplemente usaremos la clase *DigitalOut* con la que podremos definir los leds como salidas digitales, el comando *wait*, que hace a nuestro programa esperar el tiempo indicado para seguir ejecutándose a partir de ahí, y lenguaje básico de C++.

```

1 #include "mbed.h" // Incluimos la libreria de el sistema operativo de Mbed donde tenemos muchas de las funciones y clases que vamos a utilizar
2
3 DigitalOut led1(LED1); //Definimos una salida digital para el led 1 de la placa
4 DigitalOut led2(LED2); //Definimos una salida digital para el led 2 de la placa
5
6 #define SLEEP_TIME          500 // (ms) Definimos una constante con el tiempo de paropadeo del led 1
7
8 int main() // Comienza la función principal del programa
9 {
10     int count = 0;
11     while (true) {
12         // Hacemos que el led 1 cambie de estado cada 0.5 segundos
13         led1 = !led1;
14         wait_ms(SLEEP_TIME); // Función para esperar en milisegundos
15         ++count;
16         // Cada 2 segundos entraremos en este if y haremos cambiar de estado al led 2
17         if (count==4){
18             led2=!led2;
19             count=0;
20         }
21     }
22 }

```

Figura 22. Código de PruebaEntorno

Una vez tenemos todo programado simplemente compilamos el código y se nos descargará un archivo .bin con el nombre del programa, en este caso *PruebaEntorno.bin*.

Simplemente arrastramos ese archivo hacia el almacenamiento ampliado que es nuestra placa de desarrollo y se cargará el programa en ella haciendo que el led 1 parpadee cada medio segundo y el led 2 cada dos.

Como se observa el proceso es bastante sencillo y fácil de aprender, así que a primera vista parece que la opción de usar Mbed como RTOS y entorno de desarrollo es buena.

También es interesante recalcar que abriendo la librería de mbed-os, en el propio entorno de programación tenemos acceso a las clases de las que nos provee la librería y a una pequeña guía que nos explica cómo funciona cada una, cosa que resulta muy útil al estar programando.

### DigitalOut Class Reference

```
#include <DigitalOut.h>
```

**Public Member Functions**

[DigitalOut](#) (PinName pin)  
Create a **DigitalOut** connected to the specified pin.

[DigitalOut](#) (PinName pin, int value)  
Create a **DigitalOut** connected to the specified pin.

void [write](#) (int value)  
Set the output, specified as 0 or 1 (int)

int [read](#) ()  
Return the output setting, represented as 0 or 1 (int)

int [is\\_connected](#) ()  
Return the output setting, represented as 0 or 1 (int)

**DigitalOut** & [operator=](#) (int value)  
A shorthand for [write\(\)](#).

[operator int](#) ()  
A shorthand for [read\(\)](#)

Figura 23. Referencia a la clase *DigitalOut* de la librería mbed-os

Como se puede observar en la figura la documentación de cada clase nos da información sobre que funciones se pueden realizar con esa clase de dato en concreto y a veces incluso nos ofrece ejemplos de uso.

# 4 DESARROLLO DEL PROGRAMA OBJETIVO

---

En este apartado vamos a comenzar con el programa que se ha escogido como ejemplo para su desarrollo en la plataforma Mbed, para ello lo primero que se hará es explorar las aplicaciones necesarias para el proyecto dentro de Mbed-OS y ver cómo se comportan para luego poder montar el programa final sin ningún problema.

Recordemos que el proyecto que habíamos elegido como ejemplo es un sistema que deberá leer un sensor de vibraciones (acelerómetro) a través de un bus CAN, que es de lo más utilizado en aplicaciones automovilísticas, y subir esta información a la nube, además de guardarla en un sistema de almacenamiento físico en la misma placa, si es posible, para tener un histórico.

## 4.1 Exploración de las aplicaciones

Lo primero que se va a hacer es explorar cada una de las APIs que vamos a necesitar y ver cómo funciona cada una de ellas. Esto va a ser muy importante a la hora de montar el programa objetivo ya que es necesario conocer muy bien el funcionamiento de cada clase para poder usarla con buenos resultados.

### 4.1.1 Comunicación puerto serie asíncrona

Para realizar una comunicación puerto serie asíncrona no necesitamos más que dos pines que serán, el pin TX para la escritura y el pin RX para la lectura, estos pines serán necesarios en cada extremo de la comunicación y deberán estar entrelazados, es decir, si queremos comunicar el dispositivo 1 con el dispositivo 2 necesitaremos conectar el puerto TX del dispositivo 1 con el RX del dispositivo 2 y el TX del 2 con el RX del 1, de esta manera conseguiremos la conexión correcta.

En el caso de Mbed la API que controla un tipo de comunicación como esta tiene el nombre de *Serial* y con ella podremos configurar nuestra placa de desarrollo para comunicarla con cualquier otro dispositivo que acepte este tipo de comunicación, en nuestro caso el propio PC.

La clase de la que nos provee esta API funciona de una manera bastante sencilla ya que para un uso estándar de este tipo de comunicación con un PC basta con definir una variable de este tipo de clase, determinando sus pines TX y RX, usar un comando para recibir los datos que le envíe el PC y otro para enviar los datos al PC e imprimirlos por pantalla, usando para ambas cosas una aplicación como *Putty*, por ejemplo.

## Serial Class Reference

```
#include <Serial.h>
```

Inherits [SerialBase](#), and [Stream](#).

### Public Member Functions

```
Serial (PinName tx, PinName rx, const char *name=NULL)
    Create a Serial port, connected to the specified transmit and receive pins.

void baud (int baudrate)
    Set the baud rate of the serial port.

void format (int bits=8, Parity parity=SerialBase::None, int stop_bits=1)
    Set the transmission format used by the serial port.

int readable ()
    Determine if there is a character available to read.

int writable ()
    Determine if there is space available to write a character.

void attach (Callback< void()> func, IrqType type=RxIrq)
    Attach a function to call whenever a serial interrupt is generated.

template<typename T >
void attach (T *obj, void(T::*method)(), IrqType type=RxIrq)
    Attach a member function to call whenever a serial interrupt is generated.

template<typename T >
void attach (T *obj, void(*method)(T *), IrqType type=RxIrq)
    Attach a member function to call whenever a serial interrupt is generated.

void send\_break ()
    Generate a break condition on the serial line.

void send\_break ()
    Generate a break condition on the serial line.

void set\_flow\_control (Flow type, PinName flow1=NC, PinName flow2=NC)
    Set the flow control type on the serial port.

int write (const uint8_t *buffer, int length, const event\_callback\_t &callback, int event=SERIAL_EVENT_TX_COMPLETE)
    Begin asynchronous write using 8bit buffer.

int write (const uint16_t *buffer, int length, const event\_callback\_t &callback, int event=SERIAL_EVENT_TX_COMPLETE)
    Begin asynchronous write using 16bit buffer.

void abort\_write ()
    Abort the on-going write transfer.

int read (uint8_t *buffer, int length, const event\_callback\_t &callback, int event=SERIAL_EVENT_RX_COMPLETE, unsigned char char_match=SERIAL_RESERVED_CHAR_MATCH)
    Begin asynchronous reading using 8bit buffer.

int read (uint16_t *buffer, int length, const event\_callback\_t &callback, int event=SERIAL_EVENT_RX_COMPLETE, unsigned char char_match=SERIAL_RESERVED_CHAR_MATCH)
    Begin asynchronous reading using 16bit buffer.

void abort\_read ()
    Abort the on-going read transfer.

int set\_dma\_usage\_tx (DMAUsage usage)
    Configure DMA usage suggestion for non-blocking TX transfers.

int set\_dma\_usage\_rx (DMAUsage usage)
    Configure DMA usage suggestion for non-blocking RX transfers.
```

Figura 24. Funciones de la clase *Serial*

### Protected Member Functions

```
virtual void lock ()
    Acquire exclusive access to this serial port.

virtual void unlock ()
    Release exclusive access to this serial port.

virtual int close ()
    Close the file.

virtual ssize_t write (const void *buffer, size_t length)
    Write the contents of a buffer to the file.

virtual ssize_t read (void *buffer, size_t length)
    Function read Reads the contents of the file into a buffer.

virtual off_t lseek (off_t offset, int whence)
    Move the file position to a given offset from a given location.

virtual int isatty ()
    Check if the handle is for a interactive terminal device.

virtual int fsync ()
    Flush any buffers associated with the FileHandle, ensuring it is up to date on disk.
```

Figura 25. Funciones protegidas de la clase *Serial*

Además de estas funciones, que nos explica la referencia de la clase que se encuentra en el propio compilador online, existen otras básicas de C++ que también usaremos como *printf*.



El ejemplo que se ha desarrollado para comprobar el funcionamiento de esta clase es el siguiente:

```

1  #include "mbed.h" // Incluimos la librería principal del RTOS
2
3  DigitalOut myled(LED1); // Salida digital para el led1
4  DigitalOut myled2(LED2); // Salida digital para el led2
5
6  Serial pc(USBTX, USBRX); // Configuración del puerto serie (Puertos TX y RX) para el uso de un USB
7
8  int main() // Inicio de la función principal del programa
9  {
10     char k ;
11     pc.printf(" Introduce un carácter: ");
12     myled = 0; // Enciende el led 1
13     myled2 = 0; // Enciende el led 2
14     //Comienza bucle infinito
15     while(1) {
16         k = pc.getc(); // Se espera a que se introduzca un carácter y se guarda en k
17         pc.printf("%c \r \n", k); // Imprimimos por pantalla el carácter introducido
18         switch(k){
19             case 'a': //Si es una 'a' cambiamos el estado del led 1
20                 myled = !myled;
21                 break;
22
23             case 'b': //Si es una 'b' cambiamos el estado del led 2
24                 myled2 = !myled2;
25                 break;
26
27             case 'c': // Si es una 'c' imprimimos por pantalla: "Hola mundo"
28                 pc.printf("Hola mundo \r \n");
29                 break;
30
31             case 'd': //Si es una 'd' imprimimos por pantalla "Adios"
32                 pc.printf("Adios \r \n");
33                 break;
34
35             default: // En el resto de los casos se imprime "Caracter incorrecto"
36                 pc.printf("Caracter incorrecto \r \n");
37                 break;
38         }
39         pc.printf("Introduce un carácter: ");
40     }
41 }

```

Figura 26. Código del programa de prueba para la clase *Serial*

Este programa debería de ser capaz de comunicar el PC con la placa de desarrollo a través de un puerto USB, por eso los pines TX y RX son USBTX y USBRX y no unos pines convencionales, usando un protocolo de puerto serie asíncrono.

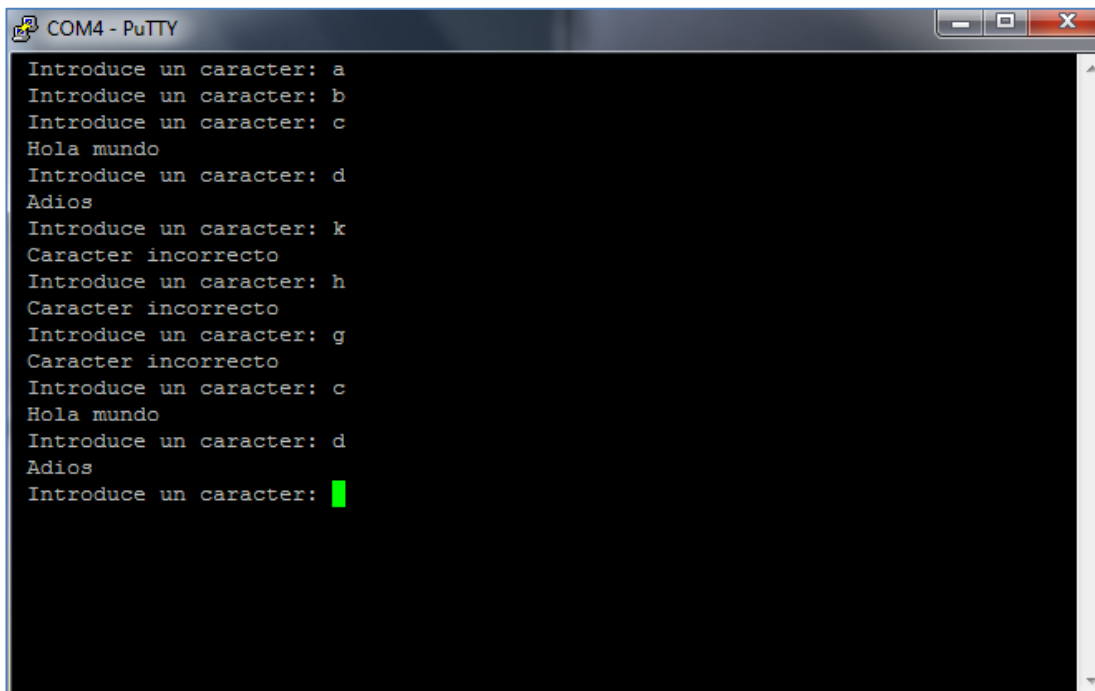
El funcionamiento del programa es sencillo, el programa nos pedirá que introduzcamos un carácter y esperará a que lo hagamos, una vez hayamos introducido el carácter este se enviará del PC a la placa y allí dependiendo de qué carácter se reciba la placa dará una respuesta u otra:

- Recibe una 'a': Cambia el estado actual del led 1 de la placa.
- Recibe una 'b': Cambia el estado actual del led 2 de la placa.
- Recibe una 'c': Envía "Hola mundo" como respuesta al PC a través del puerto serie.
- Recibe una 'd': Envía "Adiós" como respuesta al PC a través del puerto serie.
- Recibe cualquier otro carácter: Envía "Caracter incorrecto" como respuesta al PC a través del puerto serie.

Siempre devuelve por el puerto serie el carácter introducido en el PC.

Para poder probar este ejemplo será necesario trabajar con un programa como *Putty* en el PC para poder así

configurar una comunicación serie con la placa. En este caso deberemos configurar la velocidad de transmisión a 9600 Hz (o baudios) debido a que en el programa no hemos configurado ninguna y la que se aplica por defecto es esta.



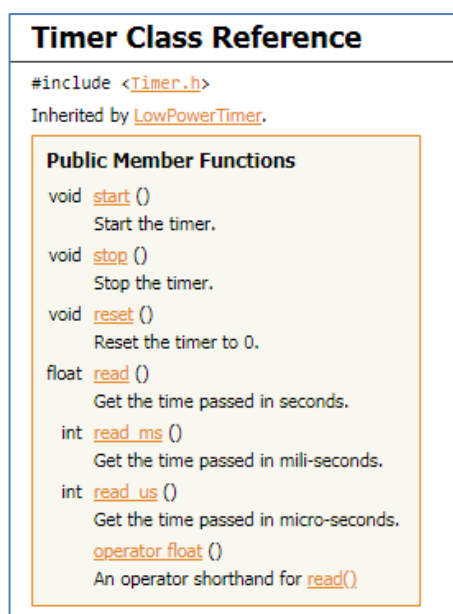
```
COM4 - PuTTY
Introduce un caracter: a
Introduce un caracter: b
Introduce un caracter: c
Hola mundo
Introduce un caracter: d
Adios
Introduce un caracter: k
Caracter incorrecto
Introduce un caracter: h
Caracter incorrecto
Introduce un caracter: g
Caracter incorrecto
Introduce un caracter: c
Hola mundo
Introduce un caracter: d
Adios
Introduce un caracter: █
```

Figura 27. Funcionamiento del programa en *Putty*

#### 4.1.2 Temporizadores y “tickers”

Estas clases pueden ser útiles en cualquier tipo de código ya que nos permiten medir el tiempo y provocar una interrupción cada cierto intervalo de tiempo.

Como ya se explicó en el apartado de las APIs los temporizadores se controlan con la API *Timer* que permite iniciar, parar y leer un temporizador con precisión mayor de milisegundo para tener controlado el tiempo que transcurre en aplicaciones que lo necesiten.



```
Timer Class Reference
#include <Timer.h>
Inherited by LowPowerTimer.

Public Member Functions
void start ()
    Start the timer.
void stop ()
    Stop the timer.
void reset ()
    Reset the timer to 0.
float read ()
    Get the time passed in seconds.
int read_ms ()
    Get the time passed in mili-seconds.
int read_us ()
    Get the time passed in micro-seconds.
operator float ()
    An operator shorthand for read().
```

Figura 28. Funciones de la clase *Timer*

En cuanto a los “tickers”, se manejan con la clase *Ticker* que ya se explicó en el apartado de APIs y que nos permite configurar una interrupción que ocurra de manera recurrente.

### Ticker Class Reference

```
#include <Ticker.h>
```

Inherits [TimerEvent](#).

Inherited by [LowPowerTicker](#), and [Timeout](#).

#### Public Member Functions

void [attach](#) (Callback< void()> func, float t)  
Attach a function to be called by the **Ticker**, specifying the interval in seconds.

template<typename T, typename M >  
void [attach](#) (T \*obj, M method, float t)  
Attach a member function to be called by the **Ticker**, specifying the interval in seconds.

void [attach\\_us](#) (Callback< void()> func, timestamp\_t t)  
Attach a function to be called by the **Ticker**, specifying the interval in micro-seconds.

template<typename T, typename M >  
void [attach\\_us](#) (T \*obj, M method, timestamp\_t t)  
Attach a member function to be called by the **Ticker**, specifying the interval in micro-seconds.

void [detach](#) ()  
Detach the function.

#### Static Public Member Functions

static void [irq](#) (uint32\_t id)  
The handler registered with the underlying timer interrupt.

Figura 29. Funciones de la clase *Ticker*

Conociendo la lista de funciones de ambas clases se va a programar una pequeña aplicación para comprobar el funcionamiento de estas APIs.

La aplicación consistirá en dos leds que parpadearán a un ritmo distinto y uno que cambiará de estado cada vez que pulsemos un interruptor, además cada vez que se pulse el interruptor se imprimirá en la pantalla del PC (usando el puerto serie USB) el tiempo que lleva activa la aplicación. El código en Mbed para esta aplicación es el siguiente:

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2
3 DigitalOut led1(LED1); // Salida digital para el led1
4 DigitalOut led2(LED2); // Salida digital para el led2
5 DigitalOut led3(LED3); // Salida digital para el led3
6
7 InterruptIn Interruptor(SW3); // Asignamos una interrupción a un pin, concretamente al interruptor 3 de la placa
8
9 Timer timer; // Creamos el temporizador
10 Ticker ticker1, ticker2; // Creamos dos tickers para dos interrupciones recurrentes
11
12 // Función 1: Cambio de estado del led 1
13 void Pled1() {
14     led1=!led1;
15 }
16 // Función 2: Cambio de estado del led 2
17 void Pled2() {
18     led2=!led2;
19 }
20 // Función 3: Cambio de estado del led 3 y impresión en pantalla del tiempo pasado desde que se inicio el temporizador
21 void Pled3() {
22     led3=!led3;
23     printf("Tiempo desde que comenzo la cuenta: %F \r \n",timer.read());
24 }
25
26 // Función principal
27 int main()
28 {
29     timer.start(); // Inicialización del timer
30     Interruptor.fall(&Pled3); //Llamada de la función 3 cuando se detecte un flanco de bajada den Interruptor (interruptor 3 de la placa)
31     ticker1.attach(&Pled1,1); //Llamada de la función 1 cada segundo
32     ticker2.attach(&Pled2,2); //Llamada de la funcion 2 cada 2 segundos
33
34 }

```

Figura 30. Código del programa de prueba para las clases *Timer* y *Ticker*

Tras compilar y cargar el binario obtenido en la placa de desarrollo se observa como los leds 1 y 2 parpadean cada segundo y cada dos segundos respectivamente. Y si pulsamos el SW3 cambia de estado el led 3 y además obtenemos por pantalla el tiempo que lleva activo el temporizador como se ve en la siguiente imagen.

```
COM4 - PuTTY
Tiempo desde que comenzo la cuenta: 6.977380
Tiempo desde que comenzo la cuenta: 27.106392
Tiempo desde que comenzo la cuenta: 37.883305
Tiempo desde que comenzo la cuenta: 42.132458
Tiempo desde que comenzo la cuenta: 44.814659
█
```

Figura 31. Funcionamiento del programa en *Putty*

Es importante resaltar que, como se puede observar en este código, se puede usar el puerto serie USB sin necesidad de inicializar una clase *Serial*, sino que un simple *printf* ya inicializa ese puerto serie y manda los datos al PC.

#### 4.1.3 Comunicación con protocolo SPI

Para este apartado se va a explicar primero a grandes rasgos cómo funciona el protocolo SPI (Serial Peripheral Interface).

El SPI es un protocolo de comunicación en serie síncrono que permite mediante una configuración maestro-esclavo un intercambio de información entre dos o más dispositivos. Este estándar incluye una línea de reloj, una línea para el dato entrante, otra para el dato saliente y un pin de selección de chip que permite seleccionar con que esclavo se comunica el maestro.

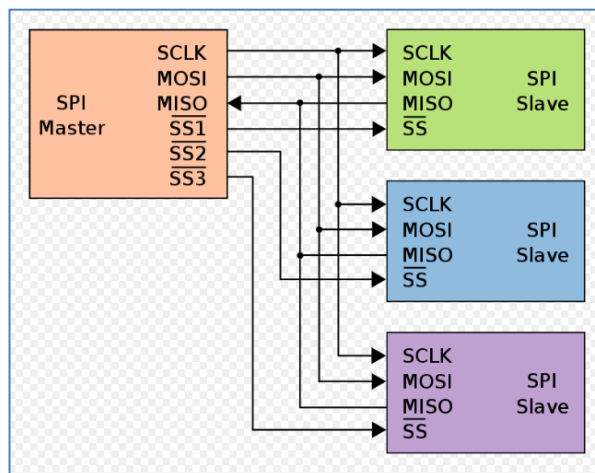


Figura 32. Esquema de protocolo SPI

Como se ve en el esquema con los terminales SS se elige con que esclavo se comunica el maestro, la línea SCLK lleva la señal de reloj que sincroniza la comunicación, los puertos MOSI son la salida de datos del maestro y entrada de datos de los esclavos (Master Out Slave In) y los puertos MISO son la salida de datos de los esclavos y entrada de datos del maestro (Master In Slave Out).

En Mbed este protocolo de comunicación se controla mediante las APIs llamadas *SPI* y *SPISlave* que controlan al maestro y al esclavo respectivamente.

Para la prueba que se va a realizar en este apartado solo necesitaremos la clase *SPI*, pero es importante conocer ambas para aplicaciones más avanzadas.

## SPI Class Reference

```
#include <SPI.h>
```

### Public Member Functions

```

SPI (PinName mosi, PinName miso, PinName sclk, PinName ssel=NC)
    Create a SPI master connected to the specified pins.

void format (int bits, int mode=0)
    Configure the data transmission format.

void frequency (int hz=1000000)
    Set the spi bus clock frequency.

virtual int write (int value)
    Write to the SPI Slave and return the response.

virtual void lock (void)
    Acquire exclusive access to this SPI bus.

virtual void unlock (void)
    Release exclusive access to this SPI bus.

template<typename Type >
    int transfer (const Type *tx_buffer, int tx_length, Type *rx_buffer, int rx_length, const event_callback_t &callback, int event=SPI_EVENT_COMPLETE)
        Start non-blocking SPI transfer using 8bit buffers.

void abort_transfer ()
    Abort the on-going SPI transfer, and continue with transfer's in the queue if any.

void clear_transfer_buffer ()
    Clear the transaction buffer.

void abort_all_transfers ()
    Clear the transaction buffer and abort on-going transfer.

int set_dma_usage (DMAUsage usage)
    Configure DMA usage suggestion for non-blocking transfers.

```

### Protected Member Functions

```

void irq_handler_async (void)
    SPI IRQ handler.

int transfer (const void *tx_buffer, int tx_length, void *rx_buffer, int rx_length, unsigned char bit_width, const event_callback_t &callback, int event)
    Common transfer method.

int queue_transfer (const void *tx_buffer, int tx_length, void *rx_buffer, int rx_length, unsigned char bit_width, const event_callback_t &callback, int event)

void start_transfer (const void *tx_buffer, int tx_length, void *rx_buffer, int rx_length, unsigned char bit_width, const event_callback_t &callback, int event)
    Configures a callback, spi peripheral and initiate a new transfer.

void start_transaction (transaction_t *data)
    Start a new transaction.

void dequeue_transaction ()
    Dequeue a transaction.

```

Figura 33. Funciones de la clase *SPI*

## SPI Slave Class Reference

```
#include <SPISlave.h>
```

### Public Member Functions

[SPISlave](#) (PinName mosi, PinName miso, PinName sclk, PinName ssel)  
Create a [SPI](#) slave connected to the specified pins.

void [format](#) (int bits, int mode=0)  
Configure the data transmission format.

void [frequency](#) (int hz=1000000)  
Set the spi bus clock frequency.

int [receive](#) (void)  
Polls the [SPI](#) to see if data has been received.

int [read](#) (void)  
Retrieve data from receive buffer as slave.

void [reply](#) (int value)  
Fill the transmission buffer with the value to be written out as slave on the next received message from the master.

Figura 34. Funciones de la clase *SPISlave*

Para probar el funcionamiento de esta API vamos a utilizar el siguiente código para conseguir una aplicación que nos devuelva por puerto serie USB el valor del registro WHOAMI de al módulo SPI.

```
1 #include "mbed.h" //Incluimos la libreria básica del RTOS
2
3 SPI spi(P0_3, P0_2,P0_0 ); // Definimos el modulo SPI maestro con su respectivo: mosi, miso y sclk
4 DigitalOut cs(P0_1); // Asignamos el pin SS de SPI de la placa a una variable digital para poder seleccionar el chip
5
6 int main() {
7     // No seleccionamos el chip
8     cs = 1;
9
10    // Configuramos la comunicación SPI para 8 bits y con estado estable de reloj a nivel alto
11    spi.format(8,3);
12    // Asignamos una frecuencia de reloj al SPI de 1MHz
13    spi.frequency(1000000);
14
15    // Seleccionamos el chip
16    cs = 0;
17
18    // Mandamos un 0x8f, el comando para leer el registro WHOAMI
19    spi.write(0x8F);
20
21    // Mandamos un byte dummy para recibir el contenido del registro WHOAMI
22    int whoami = spi.write(0x00);
23    printf("Registro WHOAMI = 0x%X\n", whoami);
24
25    // Deseleccionamos el chip
26    cs = 1;
27 }
```

Figura 35. Código del programa de prueba para la clase *SPI*

En el puerto serie recibiremos el valor del registro WHOAMI:

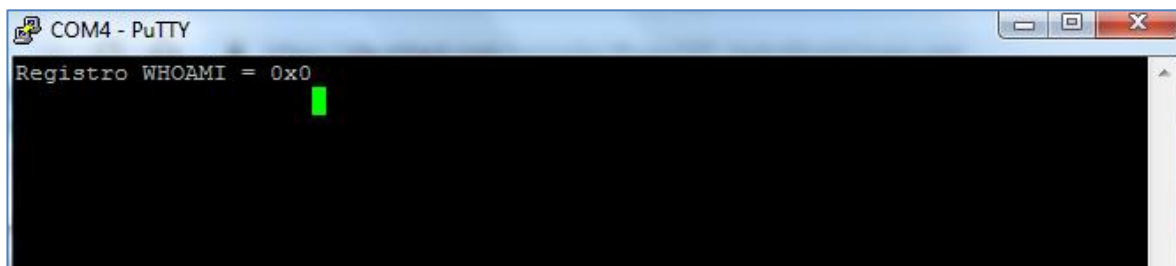


Figura 36. Respuesta del programa en *Putty*

#### 4.1.4 Conexión a un servidor TCP

Para realizar una comunicación a través de internet Mbed facilita mucho el trabajo ya que gracias a sus APIs con muy pocas líneas de código es posible conectarse a un servidor mediante el protocolo TCP o UDP y enviar o recibir datos de él. Esto podría decirse que es la parte más importante de una aplicación IoT ya que nos permite comunicarnos con la nube.

En este apartado se va a explicar cómo realizar una conexión con un servidor mediante el protocolo TCP y con conexión Ethernet que es de la que dispone la placa de pruebas en la que se está desarrollando el proyecto, aunque para usar una conexión inalámbrica solo habría que cambiar la API de Ethernet por la de Wi-Fi o la que se vaya a usar.

Las APIs que manejan esta conexión son *EthernetInterface* y *TCP Socket* que nos aportarán las clases de datos del mismo nombre y que disponen de las funciones que se muestran a continuación.

#### EthernetInterface Class Reference

```
#include <EthernetInterface.h>
```

Inherits [EthInterface](#).

##### Public Member Functions

```
virtual int connect ()
    Start the interface.

virtual int disconnect ()
    Stop the interface.

virtual const char * get\_ip\_address ()
    Get the internally stored IP address.

virtual const char * get\_mac\_address ()
    Get the internally stored MAC address.
```

##### Protected Member Functions

```
virtual NetworkStack * get\_stack ()
    Provide access to the underlying stack.
```

Figura 37. Funciones de la clase *EthernetInterface*

#### TCP Socket Class Reference

```
#include <TCP Socket.h>
```

Inherits [Socket](#).

##### Public Member Functions

```
TCP Socket ()
    Create an uninitialized socket.

template<typename S >
TCP Socket (S *stack)
    Create a socket on a network interface.

virtual ~TCP Socket ()
    Destroy a socket.

int connect (const char *host, uint16_t port)
    Connects TCP socket to a remote host.

int connect (const SocketAddress &address)
    Connects TCP socket to a remote host.

int send (const void *data, unsigned size)
    Send data over a TCP socket.

int recv (void *data, unsigned size)
    Receive data over a TCP socket.

int open (NetworkStack *stack)
    Opens a socket.

int close ()

int bind (const char *address, uint16_t port)
    Bind a specific address to a socket.

int bind (const SocketAddress &address)
    Bind a specific address to a socket.

void set\_blocking (bool blocking)
    Set blocking or non-blocking mode of the socket.

void set\_timeout (int timeout)
    Set timeout on blocking socket operations.

void attach (mbed::Callback< void()> func)
    Register a callback on state change of the socket.

template<typename T, typename M >
void attach (T *obj, M method)
    Register a callback on state change of the socket.
```

Figura 38. Funciones de la clase *TCP Socket*

Con estas APIs podemos establecer una conexión Ethernet y abrir un socket TCP para comenzar el intercambio



de información de manera muy sencilla y para comprobarlo se ha realizado un programa de prueba que enviará un mensaje sencillo al servidor y recibirá un mensaje sencillo de este.

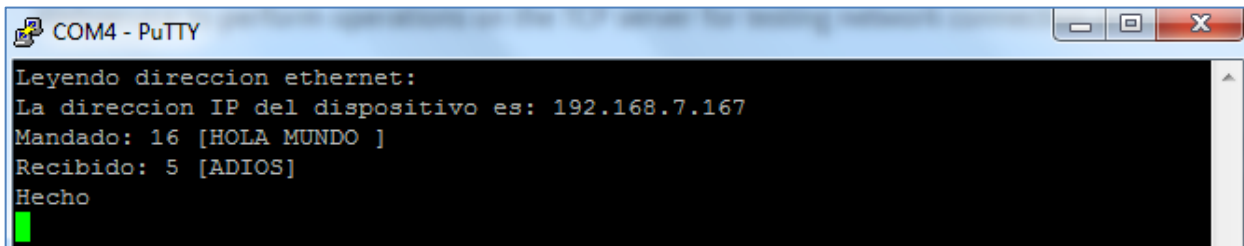
```

1  #include "mbed.h" // Incluimos la librería principal del RTOS
2  #include "EthernetInterface.h" // Incluimos la librería de para la interfaz ethernet
3
4  // Creamos una variable de clase EthernetInterface
5  EthernetInterface red;
6
7  // Programa principal
8  int main() {
9      // Conecta la interfaz ethernet
10     printf("Leyendo direccion ethernet:\r\n");
11     red.connect();
12
13     // Muestra la dirección IP (nuestra)
14     const char *ip = red.get_ip_address();
15     printf("La direccion IP del dispositivo es: %s\r\n", ip ? ip : "No IP");
16
17     // Abre un socket en la interfaz de red y crea una conexión TCP con el servidor.
18     TCPSocket socket;
19     socket.open(&red);
20     socket.connect("54.229.53.222", 8789);
21
22     // Manda un mensaje al servidor
23     char sbuffer[] = "HOLA MUNDO \r\n\r\n";
24     int scount = socket.send(sbuffer, sizeof sbuffer);
25     printf("Mandado: %d [%.*s]\r\n", scount, strlen(sbuffer), sbuffer);
26     wait(10);
27
28     // Recibe una respuesta simple y la imprime por pantalla
29     char rbuffer[64];
30     int rcount = socket.recv(rbuffer, sizeof rbuffer);
31     printf("Recibido: %d [%s]\r\n", rcount, rbuffer);
32
33     // Cierra el socket y deshabilita la interfaz de red
34     socket.close();
35
36     // Deshabilita la interfaz ethernet
37     red.disconnect();
38     printf("Hecho \r\n");

```

Figura 39. Código de la aplicación para conexión a un servidor TCP mediante ethernet

Como se puede ver en el código mandaremos un mensaje de HOLA MUNDO al servidor y este nos responderá con lo que este programado para responder.



```

COM4 - PuTTY
Leyendo direccion ethernet:
La direccion IP del dispositivo es: 192.168.7.167
Mandado: 16 [HOLA MUNDO ]
Recibido: 5 [ADIOS]
Hecho

```

Figura 40. Respuesta del programa por *Putty*

Se observa que enviamos HOLA MUNDO y recibimos diez segundos después ADIOS como respuesta (que era lo que se había programado que respondiera el servidor).

Si miramos en el servidor TCP observamos que recibe “484f4c41204d554e444f200d0a0d0a00” que si lo traducimos del hexadecimal significa “HOLA MUNDO” y enviamos “4144494F53” que traducido del



hexadecimal es “ADIOS” que como se comprueba en la consola es lo que recibimos.

```
{
  "data": "484f4c41204d554e444f200d0a0d0a00",
  "id": "8",
  "timestamp": "2019-05-31 07:58:30"
}
```

Figura 41. Mensaje que recibe el servidor

```
{
  "data": "4144494F53",
  "id": "3",
  "send": "1",
  "timestamp": "2019-05-31 07:54:38"
},
```

Figura 42. Mensaje que tiene programado el servidor para responder.

También es importante remarcar que si quisiéramos podríamos crear un servidor TCP en nuestro dispositivo utilizando la API llamada *TCPServer*.

#### 4.1.5 Conexión a un servidor UDP

El procedimiento para realizar una conexión a un servidor mediante el protocolo UDP utilizando Mbed es muy parecido al que se utiliza para realizar una conexión TCP, solo que se usará la API *UDPSocket* en lugar de la *TCPsocket*.

#### UDPSocket Class Reference

```
#include <UDPSocket.h>
```

Inherits [Socket](#).

##### Public Member Functions

```
UDPSocket ()
  Create an uninitialized socket.

template<typename S >
UDPSocket (S *stack)
  Create a socket on a network interface.

virtual ~UDPSocket ()
  Destroy a socket.

int sendto (const char *host, uint16_t port, const void *data, unsigned size)
  Send a packet over a UDP socket.

int sendto (const SocketAddress &address, const void *data, unsigned size)
  Send a packet over a UDP socket.

int recvfrom (SocketAddress *address, void *data, unsigned size)
  Receive a packet over a UDP socket.

int open (NetworkStack *stack)
  Opens a socket.

int close ()
  Close the socket.

int bind (uint16_t port)
  Bind a specific address to a socket.

int bind (const char *address, uint16_t port)
  Bind a specific address to a socket.

int bind (const SocketAddress &address)
  Bind a specific address to a socket.

void set_blocking (bool blocking)
  Set blocking or non-blocking mode of the socket.

void set_timeout (int timeout)
  Set timeout on blocking socket operations.

void attach (mbed::Callback< void()> func)
  Register a callback on state change of the socket.

template<typename T, typename M >
void attach (T *obj, M method)
  Register a callback on state change of the socket.
```

Figura 43. Funciones de la clase *UDPSocket*

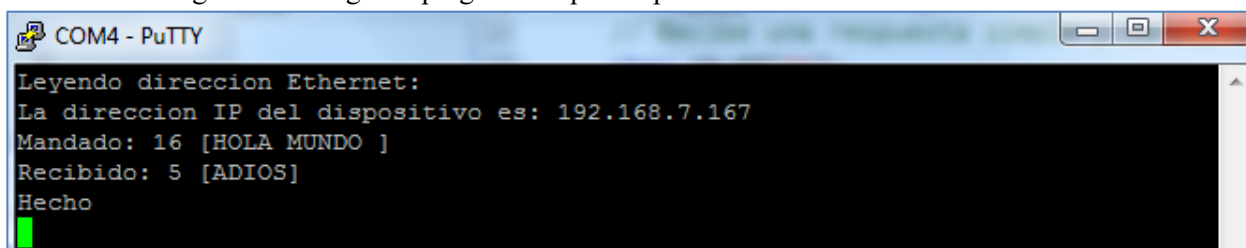
Para comprobar el funcionamiento de esta clase se va a realizar un programa de prueba que enviará un mensaje sencillo al servidor (“HOLA MUNDO”) y recibirá del otro mensaje sencillo (“ADIOS”).

```

1  #include "mbed.h" //Incluimos la libreria principal del RTOS
2  #include "EthernetInterface.h" // Incluimos la libreria de para la interfaz ethernet
3
4  // Creamos una variable de clase EthernetInterface
5  EthernetInterface red;
6
7  // Programa principal
8  int main() {
9      // Conecta la interfaz ethernet
10     printf("Leyendo direccion Ethernet:\r\n");
11     red.connect();
12
13     // Muestra la dirección IP (nuestra)
14     const char *ip = red.get_ip_address();
15     printf("La direccion IP del dispositivo es: %s\r\n", ip ? ip : "No IP");
16
17     //Abre un socket en la interfaz de red y crea una conexión UDP con el servidor
18     UDPSocket socket;
19     socket.open(&red);
20     SocketAddress dir("54.229.53.222", 8885);
21
22     // Manda un mensaje
23     char sbuffer[] = "HOLA MUNDO \r\n\r\n";
24     int scount = socket.sendto(dir, sbuffer, sizeof sbuffer);
25     printf("Mandado: %d [%s]\r\n", scount, strstr(sbuffer, "\r\n")-sbuffer, sbuffer);
26     wait(3);
27
28     // Recibe una respuesta simple y la imprime por pantalla
29     char rbuffer[64];
30     int rcount = socket.recvfrom(&dir, rbuffer, sizeof rbuffer);
31     printf("Recibido: %d [%s]\r\n", rcount, rbuffer);
32
33     // Cierra el socket y deshabilita la interfaz de red
34     socket.close();
35
36     // Deshabilita la interfaz ethernet
37     red.disconnect();
38     printf("Hecho\r\n");
39 }

```

Figura 44. Código del programa de prueba para conexión con un servidor UDP



```

COM4 - PuTTY
Leyendo direccion Ethernet:
La direccion IP del dispositivo es: 192.168.7.167
Mandado: 16 [HOLA MUNDO ]
Recibido: 5 [ADIOS]
Hecho

```

Figura 45. Respuesta del programa en *Putty*

```

{
  "data": "484f4c41204d554e444f200d0a0d0a00",
  "id": "22",
  "timestamp": "2019-05-31 09:11:24"
}

```

Figura 46. Mensaje recibido por el servidor UDP

```

{
  "data": "4144494F53",
  "id": "4",
  "send": "1",
  "timestamp": "2019-05-31 09:10:47"
}

```

Figura 47. Mensaje con el que responde el servidor

Como se puede observar el programa funciona perfectamente y su complejidad es mínima.

Como conclusión, tanto de la comunicación con un servidor TCP o con uno UDP, podemos decir que es extremadamente sencillo realizar una conexión con ellos e intercambiar datos utilizando Mbed por lo que parece que será muy útil para las aplicaciones IoT ya que se basan en esto.

#### 4.1.6 Lectura de un acelerómetro

Para leer un sensor más o menos complejo como un acelerómetro podremos hacer uso de librerías que no forman parte de las librerías básicas de Mbed pero que podemos encontrar en la biblioteca de Mbed disponible en su web.

Para nuestra prueba se ha utilizado el acelerómetro NXP MMA8652FCR1 que viene interno en la propia placa LPCXpresso54608 y para su manejo se ha descargado la librería que lo controla de la biblioteca de Mbed.

Esta librería llamada “MMA8642” nos da acceso a una nueva clase con el mismo nombre y que nos va a permitir controlar la lectura del acelerómetro de manera sencilla.

### MMA8652 Class Reference

```
#include <MMA8652.h>
```

#### Public Member Functions

[MMA8652](#) (PinName sda, PinName scl)

**MMA8652** constructor.

[~MMA8652](#) ()

**MMA8652** destructor.

void [ReadXYZ](#) (float \*a)

Get XYZ axis acceleration in floating point G's.

void [ReadXYZraw](#) (int16\_t \*d)

Get XYZ axis acceleration, signed 16 bit values.

char [getWhoAmI](#) (void)

Get the value of the WHO\_AM\_I register.

Figura 48. Lista de funciones de la clase *MMA8652*

Como podemos apreciar en la figura esta librería nos permite leer el acelerómetro con un solo comando, así que basta con que se defina una variable de tipo MMA8652 con sus pines de datos y de reloj y con un comando se podrá leer de él cuando se necesite.

En el programa de prueba leeremos el acelerómetro y según la información que obtengamos de él, encenderemos unos leds u otros y mandaremos por el puerto serie las lecturas para poder verlas en pantalla (*Putty*).

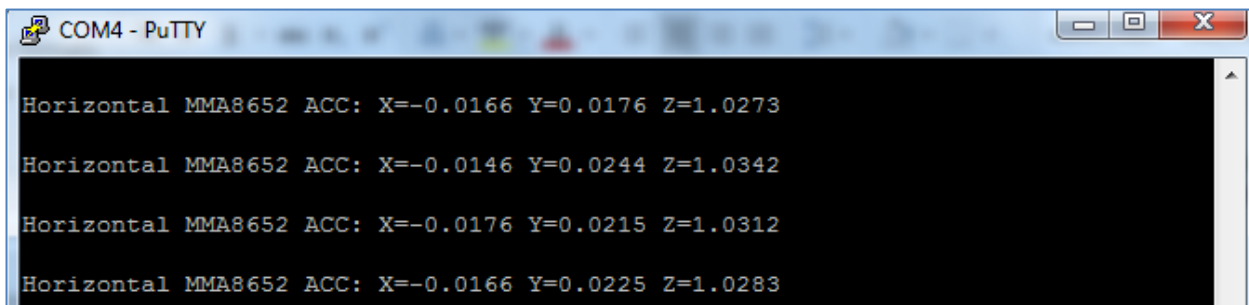
A continuación, veremos el código del programa descrito y el funcionamiento del mismo:

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2 #include "MMA8652.h" //Incluimos la librería para el acelerometro MMA8652
3
4 MMA8652 acc( D14, D15); //Creamos una variable de tipo acelerometro con sus pines sda y scl
5 Serial pc(USBTX, USBRX); //Definios una variable del tipo Serial para la comunicación puerto sere con el PC
6
7 DigitalOut led1(LED1); // Definimos una salida digital para el led 1
8 DigitalOut led2(LED2); // Definimos una salida digital para el led 2
9 DigitalOut led3(LED3); // Definimos una salida digital para el led 3
10
11 //Programa principal
12 int main()
13 {
14     float acc_data[3]; // Definimos una variable para almacenar los datos que leamos del acelerometro
15     float threshold = 0.80; //Definimos un límite superior
16     float threshold2 = 0.20; //Definimos un límite inferior
17
18     pc.printf("\r\n\r\nMMA8652 Who Am I= %X\r\n", acc.getWhoAmI()); //Obtenemos el WHOAMI del acelerometro
19     // Bucle infinito
20     while (true) {
21         acc.ReadXYZ(acc_data); //Leemos el acelerometro
22         // Si la placa esta horizontal apagamos los 3 leds
23         if (acc_data[0]<=threshold2 && acc_data[1]<=threshold && acc_data[2]>=threshold) {
24             pc.printf("Horizontal ");
25             led1 = 1;
26             led2 = 1;
27             led3 = 1;
28         }
29         // Si la placa se gira a la izquierda se encenderan los leds 2 y 3
30         else if (acc_data[0]<=threshold2 && acc_data[1]>=threshold && acc_data[2]<=threshold2) {
31             pc.printf("Izquierda ");
32             led1 = 1;
33             led2 = 0;
34             led3 = 0;
35         }
36
37         // Si la placa se gira a la derecha se encenderán los leds 1 y 3
38         else if (acc_data[0]<=threshold2 && acc_data[1]<=threshold && acc_data[2]<=threshold2) {
39             pc.printf("Derecha ");
40             led1 = 0;
41             led2 = 1;
42             led3 = 0;
43         }
44
45         //Imprimimos los datos leidos por el acelerometro cada segundo
46         pc.printf("MMA8652 ACC: X=%1.4f Y=%1.4f Z=%1.4f\r\n\r\n", acc_data[0], acc_data[1], acc_data[2]);
47         wait(1.0);
48     }
49 }
50 }

```

Figura 49. Código del programa de prueba para lectura del acelerómetro MMA8652

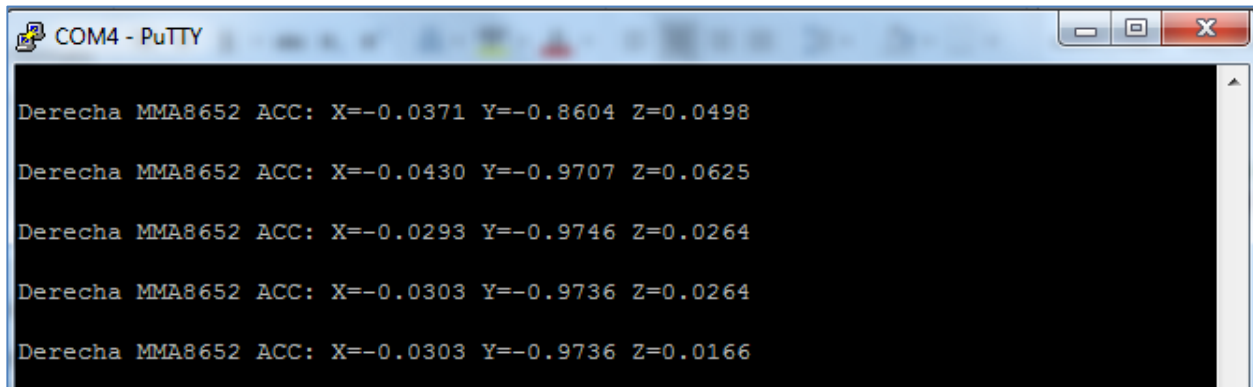


```

COM4 - PuTTY
Horizontal MMA8652 ACC: X=-0.0166 Y=0.0176 Z=1.0273
Horizontal MMA8652 ACC: X=-0.0146 Y=0.0244 Z=1.0342
Horizontal MMA8652 ACC: X=-0.0176 Y=0.0215 Z=1.0312
Horizontal MMA8652 ACC: X=-0.0166 Y=0.0225 Z=1.0283

```

Figura 50. Lecturas del acelerómetro cuando la placa está en horizontal

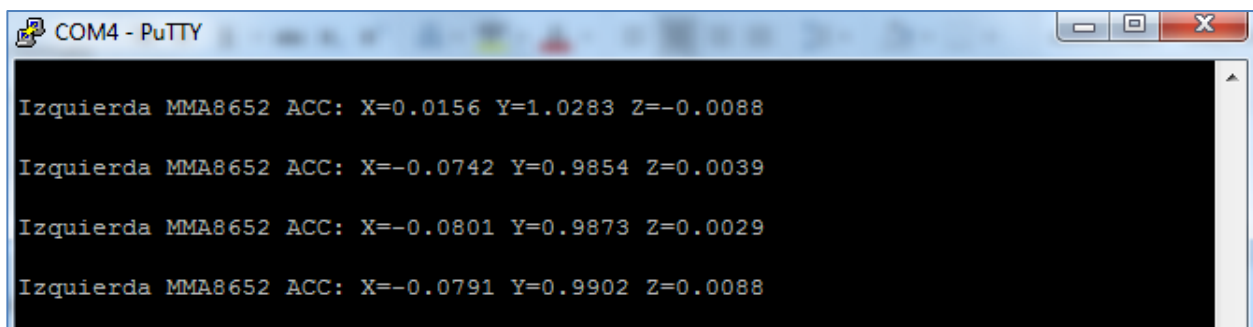


```

Derecha MMA8652 ACC: X=-0.0371 Y=-0.8604 Z=0.0498
Derecha MMA8652 ACC: X=-0.0430 Y=-0.9707 Z=0.0625
Derecha MMA8652 ACC: X=-0.0293 Y=-0.9746 Z=0.0264
Derecha MMA8652 ACC: X=-0.0303 Y=-0.9736 Z=0.0264
Derecha MMA8652 ACC: X=-0.0303 Y=-0.9736 Z=0.0166

```

Figura 51. Lecturas del acelerómetro cuando giramos la placa hacia la derecha



```

Izquierda MMA8652 ACC: X=0.0156 Y=1.0283 Z=-0.0088
Izquierda MMA8652 ACC: X=-0.0742 Y=0.9854 Z=0.0039
Izquierda MMA8652 ACC: X=-0.0801 Y=0.9873 Z=0.0029
Izquierda MMA8652 ACC: X=-0.0791 Y=0.9902 Z=0.0088

```

Figura 52. Lecturas del acelerómetro cuando giramos la placa hacia la izquierda

Además, los leds se encienden adecuadamente en cada estado.

Como vemos si tenemos disponible en la biblioteca de Mbed la librería que controla alguno de nuestros sensores el uso de estos se hace muy sencillo.

#### 4.1.7 Comunicación por medio de un bus CAN

Esta será una de las funcionalidades más importantes en el proyecto elegido, ya que básicamente necesitaremos leer de un bus CAN para recibir los datos que este introduciendo en él nuestro acelerómetro.

Para controlar un bus CAN en Mbed se pueden usar las APIs llamadas *CAN* y *CANMessage*, que nos proporcionarán una forma sencilla de comunicarnos a través de este tipo de bus.

### CANMessage Class Reference

```
#include <CAN.h>
```

Inherits CAN\_Message.

#### Public Member Functions

[CANMessage \(\)](#)

Creates empty [CAN](#) message.

[CANMessage \(int \\_id, const char \\*\\_data, char \\_len=8, CANType \\_type=CANData, CANFormat \\_format=CANStandard\)](#)

Creates [CAN](#) message with specific content.

[CANMessage \(int \\_id, CANFormat \\_format=CANStandard\)](#)

Creates [CAN](#) remote message.

Figura 53. Funciones de la clase *CANMessage*

## CAN Class Reference

```
#include <CAN.h>

Public Member Functions

    CAN (PinName rd, PinName td)
        Creates an CAN interface connected to specific pins.

    int frequency (int hz)
        Set the frequency of the CAN interface.

    int write (CANMessage msg)
        Write a CANMessage to the bus.

    int read (CANMessage &msg, int handle=0)
        Read a CANMessage from the bus.

    void reset ()
        Reset CAN interface.

    void monitor (bool silent)
        Puts or removes the CAN interface into silent monitoring mode.

    int mode (Mode mode)
        Change CAN operation to the specified mode.

    int filter (unsigned int id, unsigned int mask, CANFormat format=CANAny, int handle=0)
        Filter out incoming messages.

    unsigned char rerror ()
        Returns number of read errors to detect read overflow errors.

    unsigned char terror ()
        Returns number of write errors to detect write overflow errors.

    void attach (Callback< void()> func, IrqType type=RxIrq)
        Attach a function to call whenever a CAN frame received interrupt is generated.

    template<typename T >
        void attach (T *obj, void(T::*method)(), IrqType type=RxIrq)
            Attach a member function to call whenever a CAN frame received interrupt is generated.

    template<typename T >
        void attach (T *obj, void(*method)(T *), IrqType type=RxIrq)
            Attach a member function to call whenever a CAN frame received interrupt is generated.
```

Figura 54. Funciones de la clase *CAN*

Como se puede observar en las figuras 53 y 54, es bastante sencillo trabajar con un bus CAN usando estas APIs ya que basta con crear un mensaje CAN y trabajar con el enviándolo o leyéndolo además de poder cambiar la velocidad de transmisión y otras opciones.

El primer problema grave del uso de Mbed que se ha encontrado, aparece cuando al compilar la prueba de uso de este tipo de comunicación el compilador online nos avisa de que es imposible compilar el código y después de muchos arreglos, cambios y consultas en los foros de Mbed se llega a la conclusión de que lo que ocurre es que nuestra placa de desarrollo LPCXpresso54608 cuenta con comunicación CAN (TX y RX), pero no está montada con el mismo estándar que usa la librería *CAN* de Mbed y por lo tanto no podemos usar esta API en nuestra placa de desarrollo.

Esto es uno de los mayores problemas de Mbed descubiertos durante este proyecto, ya que nos da a entender que Mbed funciona muy bien para la mayoría de las aplicaciones y es más cómodo que muchos otros entornos de programación al trabajar a tan alto nivel, pudiendo así reducir mucho las horas de trabajo en el desarrollo de muchas aplicaciones IoT, pero esto solo será así siempre y cuando nuestro dispositivo sea compatible con las librerías y APIs con las que cuenta Mbed, ya que cuando no lo sea deberemos recurrir a diseñarlas a bajo nivel, cosa que puede ser incluso más complicada que hacerlo en el propio entorno de programación del fabricante de la placa, debido a que allí contaremos con más librerías para esto.

Viendo este problema y teniendo acceso a la placa de desarrollo NUCLEO-L4R5ZI-P, realizaremos la prueba en esta placa para ver si la comunicación CAN de esta es compatible con las APIs para CAN de Mbed.

Al intentar realizar una prueba usando la placa de desarrollo NUCLEO-L4R5ZI-P se comprueba que este dispositivo sí que es compatible con las APIs de bus CAN de Mbed, pero surge el problema de que esta placa solo cuenta con un canal de CAN y por lo tanto no podemos enviar y recibir en la misma placa por lo que no es posible comprobar el funcionamiento del programa.

Ante estos resultados se deduce que no se va a poder realizar una prueba utilizando solo esta API ya que no

contamos con los equipos compatibles necesarios para realizarla.

#### 4.1.8 Almacenamiento de datos en una tarjeta SD

El almacenamiento de datos en una tarjeta SD puede ser muy útil en una aplicación IoT ya que nos proporcionará un histórico en el lugar donde este instalado el dispositivo al que podremos acceder “físicamente” si lo necesitamos.

En Mbed este tipo de aplicación se controla con la API llamada *SDBlockDevice* que, en teoría, permite escribir, leer, borrar, etc de una tarjeta SD de manera muy sencilla.

##### 📁 SDBlockDevice Class Reference

Public Member Functions	
	<a href="#">SDBlockDevice</a> (PinName mosi, PinName miso, PinName sclk, PinName cs, uint64_t hz=1000000, bool crc_on=0)
	Creates an <a href="#">SDBlockDevice</a> on a SPI bus specified by pins. <a href="#">More...</a>
virtual int	<a href="#">init</a> ()
	Initialize a block device. <a href="#">More...</a>
virtual int	<a href="#">deinit</a> ()
	Deinitialize a block device. <a href="#">More...</a>
virtual int	<a href="#">read</a> (void *buffer, mbed::bd_addr_t addr, mbed::bd_size_t size)
	Read blocks from a block device. <a href="#">More...</a>
virtual int	<a href="#">program</a> (const void *buffer, mbed::bd_addr_t addr, mbed::bd_size_t size)
	Program blocks to a block device. <a href="#">More...</a>
virtual int	<a href="#">trim</a> (mbed::bd_addr_t addr, mbed::bd_size_t size)
	Mark blocks as no longer in use. <a href="#">More...</a>
virtual mbed::bd_size_t	<a href="#">get_read_size</a> () const
	Get the size of a readable block. <a href="#">More...</a>
virtual mbed::bd_size_t	<a href="#">get_program_size</a> () const
	Get the size of a programmable block. <a href="#">More...</a>
virtual mbed::bd_size_t	<a href="#">size</a> () const
	Get the total size of the underlying device. <a href="#">More...</a>
virtual void	<a href="#">debug</a> (bool dbg)
	Enable or disable debugging. <a href="#">More...</a>
virtual int	<a href="#">frequency</a> (uint64_t freq)
	Set the transfer frequency. <a href="#">More...</a>
virtual const char *	<a href="#">get_type</a> () const
	Get the BlockDevice class type. <a href="#">More...</a>
virtual int	<a href="#">sync</a> ()
	Ensure data on storage is in sync with the driver. <a href="#">More...</a>



virtual int	<code>erase (bd_addr_t addr, bd_size_t size)</code>
	Erase blocks on a block device. <a href="#">More...</a>
virtual bd_size_t	<code>get_erase_size () const</code>
	Get the size of an erasable block. <a href="#">More...</a>
virtual bd_size_t	<code>get_erase_size (bd_addr_t addr) const</code>
	Get the size of an erasable block given address. <a href="#">More...</a>
virtual int	<code>get_erase_value () const</code>
	Get the value of storage when erased. <a href="#">More...</a>
virtual bool	<code>is_valid_read (bd_addr_t addr, bd_size_t size) const</code>
	Convenience function for checking block read validity. <a href="#">More...</a>
virtual bool	<code>is_valid_program (bd_addr_t addr, bd_size_t size) const</code>
	Convenience function for checking block program validity. <a href="#">More...</a>
virtual bool	<code>is_valid_erase (bd_addr_t addr, bd_size_t size) const</code>
	Convenience function for checking block erase validity. <a href="#">More...</a>

Figura 55. Funciones de la clase *SDBlockDevice*

El problema aparece de nuevo cuando al intentar utilizar esta API en nuestra placa de desarrollo LPCXpresso54608 el compilador online no deja compilar el código debido a que la memoria SD de nuestra placa no es compatible con las APIs de control para este tipo de memoria que trae consigo Mbed.

De nuevo vemos el problema de que sería muy sencillo trabajar utilizando esta API con memorias SD, pero si nuestro dispositivo no es compatible con ellas (cosa que no es difícil que ocurra) perdemos toda la capacidad de mejora que nos ofrecía Mbed a la hora de trabajar.

En este caso ni siquiera podemos intentar diseñar esta función para la placa de desarrollo NUCLEO-L4R5ZI-P, que parece ser más compatible con Mbed, debido a que esta placa no cuenta con lector de tarjetas SD.

Debido a todo esto dejaremos esta funcionalidad de lado para nuestro ejemplo de uso final.

## 4.2 Acoplamiento de funcionalidades

En este apartado se realizarán algunos programas que unirán algunas de las funcionalidades exploradas en el apartado anterior, con el objetivo de observar cómo se acoplan estas funcionalidades en un mismo programa y allanar el terreno para el diseño del ejemplo final con el que se cerrará este proyecto sobre el uso Mbed en aplicaciones IoT.

### 4.2.1 Lectura de acelerómetro y envío de datos a un servidor TCP

En este apartado se va a unir la funcionalidad de leer un acelerómetro y la de escribir en un servidor TCP, que será básicamente lo que queremos que consiga nuestro sistema final, para ello solamente hará falta utilizar las APIs *MMA8642*, *EthernetInterface* y *TCP Socket* de Mbed exploradas en el apartado anterior.

El objetivo del programa será leer los datos del acelerómetro cada cierto tiempo y subirlos al servidor TCP, pero también se ha desarrollado un ejemplo en el que se suben los datos al servidor cada vez que se introduce una 'a' por puerto serie al dispositivo.

Empecemos por la funcionalidad más útil: subir los datos del acelerómetro al servidor cada 5 segundos.



```

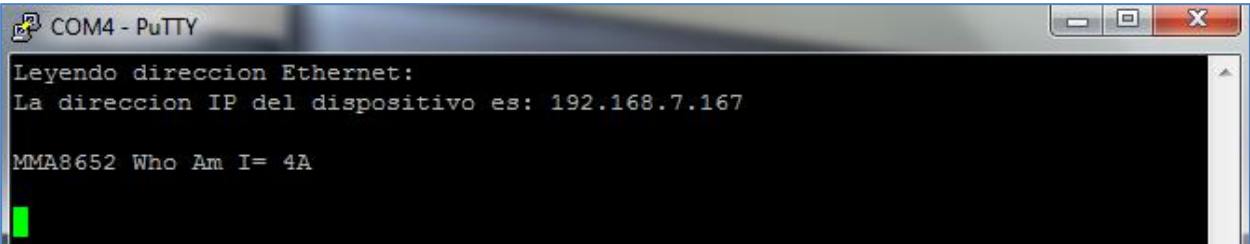
1 #include "mbed.h" // Incluimos la librería principal del RTOS
2 #include "MMA8652.h" // Incluimos la librería de control del acelerometro
3 #include "EthernetInterface.h" // Incluimos la librería para la interfaz ethernet
4
5 MMA8652 acc( D14, D15); // Creamos la variable para el acelerometro
6
7 EthernetInterface red; // Creamos la variable para la conexión ethernet
8
9 Serial pc(USBTX, USBRX); // Definimos un puerto serie USB
10
11 //Programa principal
12 int main()
13 {
14     // Configuramos la interfaz Ethernet e imprimimos por pantalla nuestra dirección IP.
15     printf("Leyendo direccion Ethernet:\r\n");
16     red.connect();
17     const char *ip = red.get_ip_address();
18     printf("La direccion IP del dispositivo es: %s\r\n", ip ? ip : "No IP");
19
20     //Inicializamos el acelerometro y creamos el vector x, y, z.
21     printf("\r\nMMA8652 Who Am I= %X\r\n", acc.getWhoAmI());
22     float acc_data[3];
23     //Esperamos una 'a' por el puerto serie para iniciar la transmisión
24     char k = pc.getc();
25     if(k=='a') {
26         while (1) {
27             //Leemos el acelerometro en acc_data e imprimimos por pantalla los valores que obtenemos en cada eje.
28             acc.ReadXYZ(acc_data);
29             printf("MMA8652 ACC: X=%1.4f Y=%1.4f Z=%1.4f\r\n", acc_data[0], acc_data[1], acc_data[2]);
30
31             red.connect();
32             const char *ip = red.get_ip_address();
33             //Configuramos el socket TCP
34             TCPSocket socket; //Creamos un socket TCP
35             socket.open(&red);
36             socket.connect("54.229.53.222", 8789);
37
38             //Mandamos la información al servidor
39             socket.send(acc_data,sizeof acc_data);
40
41             //Imprimimos en pantalla la información enviada
42             printf("Enviado en x: [%1.4f]\r\n", acc_data[0]);
43             printf("Enviado en y: [%1.4f]\r\n", acc_data[1]);
44             printf("Enviado en z: [%1.4f]\r\n", acc_data[2]);
45
46             //Esperamos 5 segundos hasta la siguiente lectura y envío de datos
47             wait(5);
48         }
49     }
50 }

```

Figura 56. Código de aplicación para envío de datos del acelerometro a un servidor TCP cada 5 segundos

Como se puede observar en el código, cuando ejecutemos el programa este nos informará de la dirección IP del dispositivo y del WhoAmI del acelerómetro. Luego esperará una 'a' por el puerto serie para comenzar el envío de información, esto se ha incluido para que cuando haya que realizar alguna comprobación se pueda resetear el dispositivo y desconectarlo sin interrumpir directamente el envío, cuando se reciba la 'a' cada 5 segundos se leerá el acelerómetro, se informará de lo leído por el puerto serie (para pruebas de funcionamiento o mantenimiento) y se enviarán al servidor los datos que se están leyendo.

Para comprobar el funcionamiento de este código se ha conectado la placa con un cable ethernet con conexión y con el PC para la comunicación serie, luego se ha ejecutado y este es el resultado.



```

COM4 - PuTTY
Leyendo direccion Ethernet:
La direccion IP del dispositivo es: 192.168.7.167

MMA8652 Who Am I= 4A

```

Figura 57. Sistema inicializado y esperando el carácter 'a' para empezar a transmitir

```

COM4 - PuTTY
Leyendo direccion Ethernet:
La direccion IP del dispositivo es: 192.168.7.167

MMA8652 Who Am I= 4A

MMA8652 ACC: X=-0.0137 Y=0.0186 Z=1.0322
Enviado en x: [-0.0137]
Enviado en y: [0.0186]
Enviado en z: [1.0322]

MMA8652 ACC: X=-0.0117 Y=0.0186 Z=1.0312
Enviado en x: [-0.0117]
Enviado en y: [0.0186]
Enviado en z: [1.0312]

MMA8652 ACC: X=-0.0166 Y=0.0195 Z=1.0312
Enviado en x: [-0.0166]
Enviado en y: [0.0195]
Enviado en z: [1.0312]

MMA8652 ACC: X=-0.0439 Y=0.9814 Z=0.0488
Enviado en x: [-0.0439]
Enviado en y: [0.9814]
Enviado en z: [0.0488]

MMA8652 ACC: X=-0.1455 Y=-0.9482 Z=0.0576
Enviado en x: [-0.1455]
Enviado en y: [-0.9482]
Enviado en z: [0.0576]

```

Figura 58. Sistema transmitiendo al servidor TCP

A través del puerto serie podemos ver en el PC los datos que está enviando el dispositivo al servidor, ahora vamos a comprobar que los datos que llegan al servidor sean correctos. Para ello accedemos a la web de información del servidor y observamos que está recibiendo.

```

[
  {
    "data": "000060bc0000983c0020843f",
    "id": "1",
    "timestamp": "2019-06-06 09:34:29"
  },
  {
    "data": "000040bc0000983c0000843f",
    "id": "2",
    "timestamp": "2019-06-06 09:34:34"
  },
  {
    "data": "000088bc0000a03c0000843f",
    "id": "3",
    "timestamp": "2019-06-06 09:34:39"
  },
]

```

Figura 59. Tres primeros mensajes recibidos por el servidor TCP

```

{
  "data": "000034bd00407b3f0000483d",
  "id": "4",
  "timestamp": "2019-06-06 09:34:44"
},
{
  "data": "000015be00c072bf00006c3d",
  "id": "5",
  "timestamp": "2019-06-06 09:34:50"
},
]

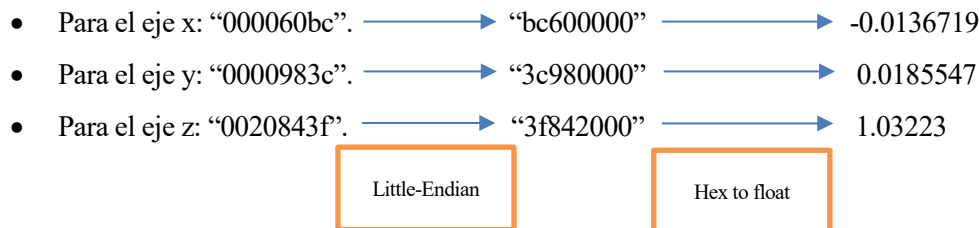
```

Figura 60. Cuarto y quinto mensajes recibidos por el servidor TCP

Como se puede observar, el servidor TCP recibe un mensaje cada 5 segundos, pero este mensaje está en hexadecimal por lo que será necesario decodificarlo para comprobar si la información es correcta.

La información se manda en float que ocupan una palabra de 4 bytes cada uno por lo que tenemos que dividir el mensaje en 3, un byte para cada eje del acelerómetro. Además, se debe tener en cuenta que la información se envía en *Little-Endian* es decir que, aunque se reciba en el servidor una palabra de cuatro bytes así “0020843f” en realidad la información que estamos enviando es “3f842000” ya que se envían los bytes menos significativos primero.

Teniendo esto en cuenta, el primer mensaje que recibe el servidor es “000060bc0000983c0020843f” que si lo dividimos en las tres palabras de cuatro bytes cada una quedan así:



Como vemos coincide completamente con el valor que se ha leído del acelerómetro, con el resto de los envíos se puede comprobar que también coinciden.

Ahora vamos a proponer una alternativa menos útil para aplicaciones IoT, ya que necesita de alguien enviando datos por el puerto serie, pero posiblemente más cómoda para realizar ensayos en un laboratorio ya que se puede controlar cuando leer y enviar la información del acelerómetro al servidor.

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2 #include "MMA8652.h" // Incluimos la librería de control del acelerometro
3 #include "EthernetInterface.h" // Incluimos la librería para la interfaz ethernet
4
5 MMA8652 acc( D14, D15); // Creamos la variable para el acelerometro
6
7 EthernetInterface red; // Creamos la variable para la conexión ethernet
8
9 Serial pc(USBTX, USBRX); // Definimos un puerto serie USB
10
11 //Programa principal
12 int main()
13 {
14     // Configuramos la interfaz Ethernet y imprimimos por pantalla nuestra dirección IP.
15     printf("Leyendo dirección ethernet: \r\n");
16     red.connect();
17     const char *ip = red.get_ip_address();
18     printf("La dirección IP del dispositivo es: %s\r\n", ip ? ip : "No IP");
19
20     //Inicializamos el acelerometro y creamos el vector x, y, z.
21     printf("\r\n\r\nMMA8652 Who Am I= %X\r\n", acc.getWhoAmI());
22     float acc_data[3];
23
24     TCPSocket socket; // Creamos el socket TCP
25
26     char k='a';
27     while (1) {
28         //Leemos el acelerometro en acc_data e imprimimos por pantalla los valores que obtenemos en cada eje.
29         acc.ReadXYZ(acc_data);
30         printf("MMA8652 ACC: X=%1.4f Y=%1.4f Z=%1.4f\r\n", acc_data[0], acc_data[1], acc_data[2]);
31
32         red.connect();
33         const char *ip = red.get_ip_address();
34         //Configuramos el socket TCP
35         TCPSocket socket; // Creamos el socket TCP
36         socket.open(&red);
37         socket.connect("54.229.53.222", 8789);
38
39         //Mandamos la información al servidor
40         socket.send(acc_data,sizeof acc_data);

```

```

40     socket.send(acc_data,sizeof acc_data);
41
42     //Imprimimos en pantalla la información enviada
43     printf("Enviado en x: [%1.4f]\r\n", acc_data[0]);
44     printf("Enviado en y: [%1.4f]\r\n", acc_data[1]);
45     printf("Enviado en z: [%1.4f]\r\n", acc_data[2]);
46
47     // Introduciendo una 'a' por el puerto serie haremos otro envío de información al servidor
48     // E introduciendo cualquier otro caracter pararemos la lectura dela acelerometro y nos desconectaremos del servidor
49     k = pc.getc();
50     if(k!='a') {
51         break;
52     }
53 }
54 //Sale del envío y se queda en paro hasta recibir un reset
55 socket.close();
56 red.disconnect();
57 printf("Desconectado \r\n");
58 }

```

Figura 61. Código de la aplicación para envío manual de datos de un acelerómetro a un servidor TCP

El código funciona correctamente y se puede comprobar su funcionamiento de la misma forma que el anterior.

## 4.2.2 Lectura de acelerómetro y envío de datos a un servidor UDP

En este apartado haremos exactamente lo mismo que en el apartado anterior, pero con un servidor UDP en vez de uno TCP.

De esta forma podremos mandar la información obtenida del acelerómetro a un servidor con ambos protocolos.

```

1  #include "mbed.h" // Incluimos la librería principal del RTOS
2  #include "MMA8652.h" // Incluimos la librería de control del acelerometro
3  #include "EthernetInterface.h" // Incluimos la librería para la interfaz ethernet
4
5  MMA8652 acc( D14, D15); // Creamos la variable para el acelerometro
6
7  EthernetInterface red; // Creamos la variable para la conexión ethernet
8
9  SocketAddress dir("54.229.53.222", 8885); // Definimos la dirección del socket UDP
10
11 Serial pc(USBTX, USBRX); // Definimos un puerto serie USB
12
13 //Programa principal
14 int main()
15 {
16     // Configuramos la interfaz Ethernet e imprimimos por pantalla nuestra dirección IP.
17     printf("Leyendo direccion Ethernet:\r\n");
18     red.connect();
19     const char *ip = red.get_ip_address();
20     printf("La direccion IP del dispositivo es: %s\r\n", ip ? ip : "No IP");
21
22     //Inicializamos el acelerometro y creamos el vector x, y, z.
23     printf("\r\nMMA8652 Who Am I= %X\r\n", acc.getWhoAmI());
24     float acc_data[3];
25
26     //Esperamos una 'a' por el puerto serie para iniciar la transmisión
27     char k = pc.getc();
28     if(k=='a') {
29         while (1) {
30
31             //Leemos el acelerometro en acc_data e imprimimos por pantalla los valores que obtenemos en cada eje.
32             acc.ReadXYZ(acc_data);
33             printf("MMA8652 ACC: X=%1.4f Y=%1.4f Z=%1.4f\r\n", acc_data[0], acc_data[1], acc_data[2]);
34
35             red.connect();
36             const char *ip = red.get_ip_address();
37
38             //Abre un socket en la interfaz de red y crea una conexión UDP con el servidor.
39             UDPSocket socket;
40             socket.open(&red);

```

```
41
42     //Mandamos la información al servidor
43     socket.sendto(dir,acc_data,sizeof acc_data);
44
45     //Imprimimos en pantalla la información enviada
46     printf("Enviado en x: [%1.4f]\r\n", acc_data[0]);
47     printf("Enviado en y: [%1.4f]\r\n", acc_data[1]);
48     printf("Enviado en z: [%1.4f]\r\n\n", acc_data[2]);
49
50     //Esperamos 5 segundos hasta la siguiente lectura y envío de datos
51     wait(5);
52 }
53 }
54 }
```

Figura 62. Código de aplicación para envío de datos del acelerómetro a un servidor UDP cada 5 segundos

El funcionamiento de esta aplicación es básicamente igual al de la comunicación del acelerómetro con el servidor TCP, primero nos informa de la dirección IP del dispositivo y el WhoAmI del acelerómetro y se queda esperando una 'a' por puerto serie y luego cuando recibe la 'a' comienza a leer el acelerómetro y enviar la información al servidor UDP cada 5 segundos.

Ahora se va a comprobar su funcionamiento:

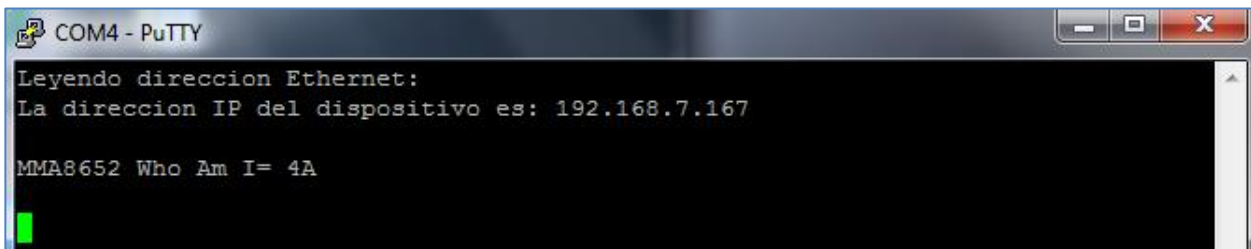


Figura 63. Sistema inicializado y esperando el carácter 'a' para empezar a transmitir

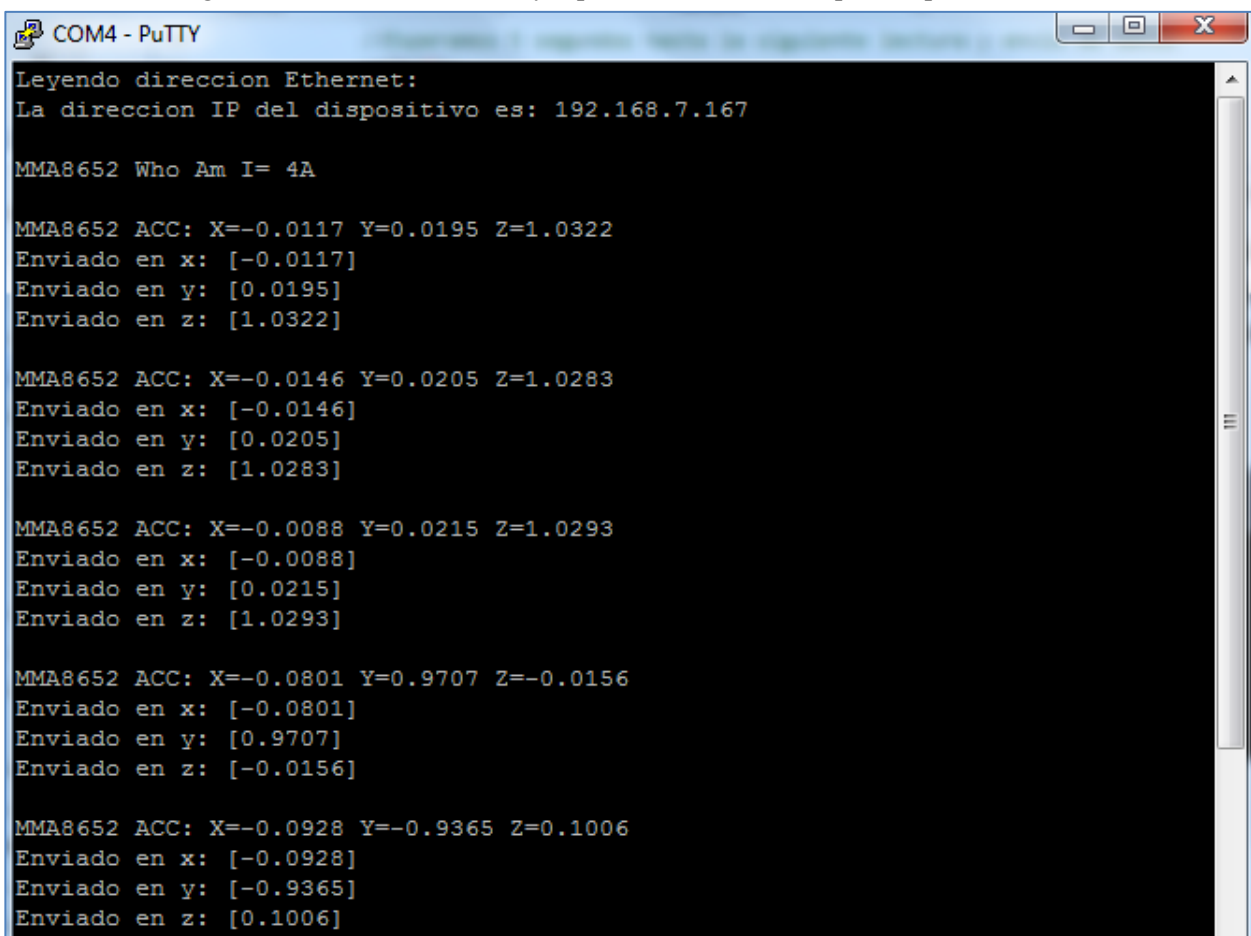


Figura 64. Sistema transmitiendo al servidor UDP

```

{
  "data": "000040bc0000a03c0020843f",
  "id": "1",
  "timestamp": "2019-06-06 11:51:37"
},
{
  "data": "000070bc0000a83c00a0833f",
  "id": "2",
  "timestamp": "2019-06-06 11:51:42"
},
{
  "data": "000010bc0000b03c00c0833f",
  "id": "3",
  "timestamp": "2019-06-06 11:51:47"
},

```

Figura 65. Tres primeros mensajes recibidos por el servidor UDP

```

{
  "data": "0000a4bd0080783f000080bc",
  "id": "4",
  "timestamp": "2019-06-06 11:51:52"
},
{
  "data": "0000bebd00c06fbf0000ce3d",
  "id": "5",
  "timestamp": "2019-06-06 11:51:57"
},

```

Figura 66. Cuarto y quinto mensajes recibidos por el servidor UDP

Con todos estos datos a mano se puede observar que el servidor recibe mensajes cada 5 segundos y que estos se corresponden con la información leída por el acelerómetro, por ejemplo el cuarto mensaje que se ha enviado desde el acelerómetro es:  $X = -0.0801$ ,  $Y = 0.9707$  y  $Z = -0.0156$ .

Si miramos en el servidor UDP observamos que se recibe el mensaje "0000a4bd0080783f000080bc" que dividido en tres palabras de cuatro bytes cada una y sabiendo que se envía en *Little-Endian* tendríamos: "bda40000" para X, "3f788000" para Y y "bc800000" para Z. Si estos valores los pasamos de hexadecimal a float obtenemos que:  $X = -0.0800781$ ,  $Y = 0.970703$  y  $Z = -0.015625$ , que como vemos coincide con lo leído por el acelerómetro por lo que se deduce que el dispositivo está funcionando correctamente.

### 4.2.3 Comunicación CAN – SPI entre dos dispositivos

En este apartado se va a intentar realizar una prueba que consistirá en interconectar la placa de desarrollo NUCLEO-L4R5ZI-P y la LPCXpresso54608 y controlar la comunicación CAN de la primera utilizando las APIs para CAN de Mbed y la comunicación CAN de la segunda desde SPI, además se utilizarán los MCP2515 y MCP2551 para montar la red de comunicación necesaria entre ambos dispositivos.

Para realizar esta prueba se deberá realizar un montaje que consiga conectar ambas placas de desarrollo como se especifica arriba y con los datos en el formato adecuado para que el dispositivo que deba leerlos los pueda entender.

Para este montaje inicial necesitaremos 16 cables, una proto-board, una resistencia de  $120\Omega$  (la otra va en el terminal del CAN-BUS Shield), un MCP2551, un CAN-BUS Shield V1.2 de DIGITAL y las placas de desarrollo LPCXpresso54608 y NUCLEO-L4R5ZI-P.

El montaje seguirá el siguiente esquema:

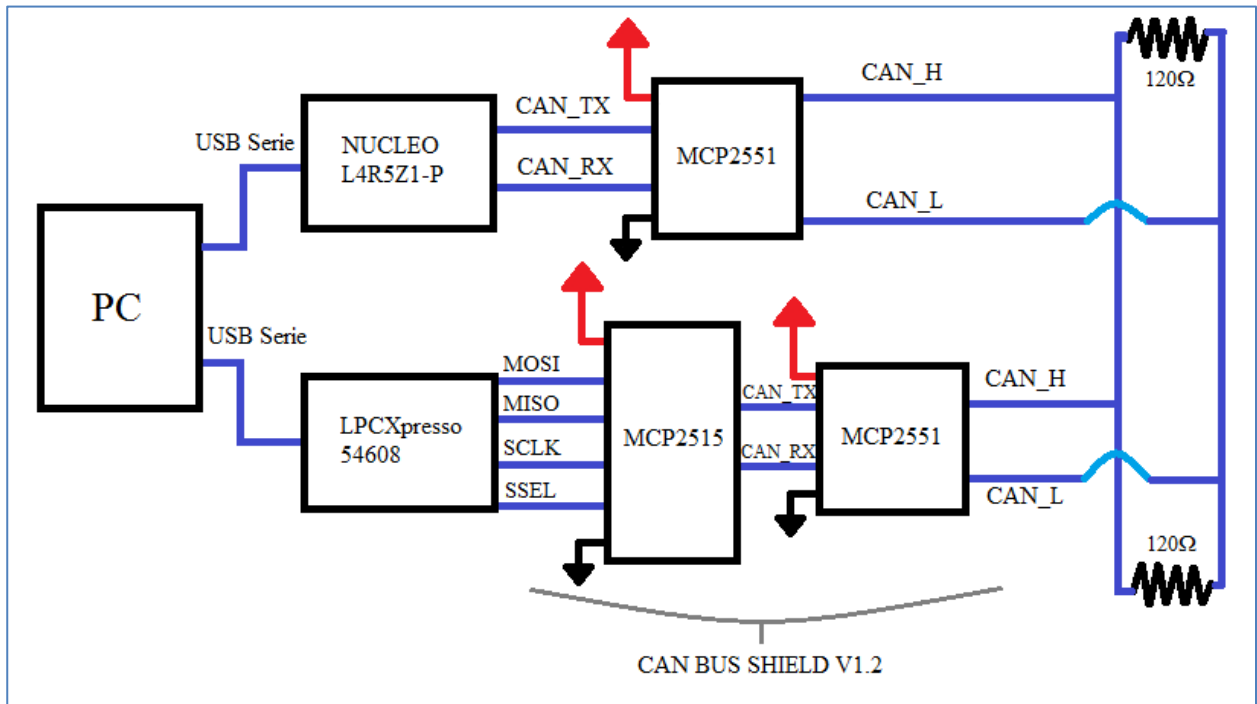


Figura 67. Esquema del montaje para comunicación CAN

Dividiremos este montaje en dos partes, la primera será desde la NUCLEO L4R5Z1-P hasta el bus CAN (CAN\_H, CAN\_L) y la segunda desde el LPCXpresso54608 hasta el bus CAN también

Montaremos la primera parte y haremos una prueba sencilla para intentar medir con un osciloscopio en el bus CAN y comprobar si el bus funciona correctamente. Una vez conseguido esto, montaremos la segunda parte y probaremos si el sistema completo funciona.

Ya que vamos a medir el bus CAN con el osciloscopio es importante comprender bien cómo funciona este físicamente por lo tanto se va a explicar un poco como funciona este tipo de comunicación.

Es necesario comprender que la comunicación por medio de bus CAN es interesante ya que nos permite conectar muchos dispositivos de entrada o salida al bus, que solo tiene dos cables (CAN H y CAN L), y de esta manera podemos tener un sistema con muchos dispositivos conectados entre sí usando solo dos cables, esto es interesante ya que reduce los costes en cuanto a cables se refiere, además reduce el peso del sistema y presenta una alta inmunidad a las interferencias por lo que para aplicaciones automovilísticas es lo más usado.

El esquema básico de un bus CAN consiste en dos unidades en los extremos de este bus con una impedancia de 120Ω cada una entre la línea de CAN H y la de CAN L estas unidades están unidas mediante los dos cables del bus y entre ellas podemos conectar al bus en paralelo los dispositivos que necesitemos.

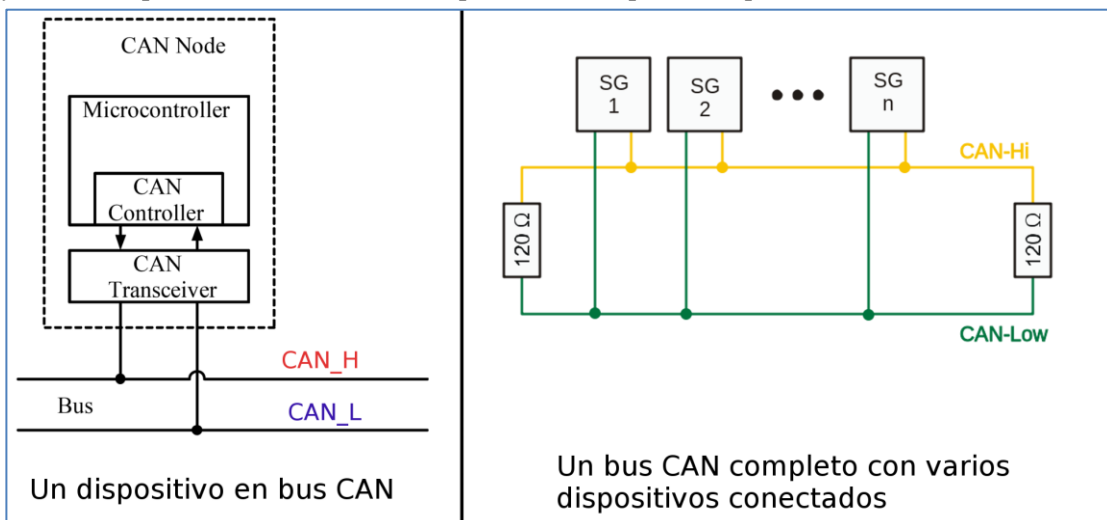


Figura 68. Esquema de bus CAN estándar



Hay que tener en cuenta los niveles de tensión a los que trabaja el bus y estos son:

- CAN H: Trabaja entre 2.5V y 3.5V
- CAN L: Trabaja entre 1.5V y 2.5V

A la hora del intercambio de información el bus trabaja en binario pero la codificación de estos bits en las tensiones de CAN H y CAN L es la siguiente:

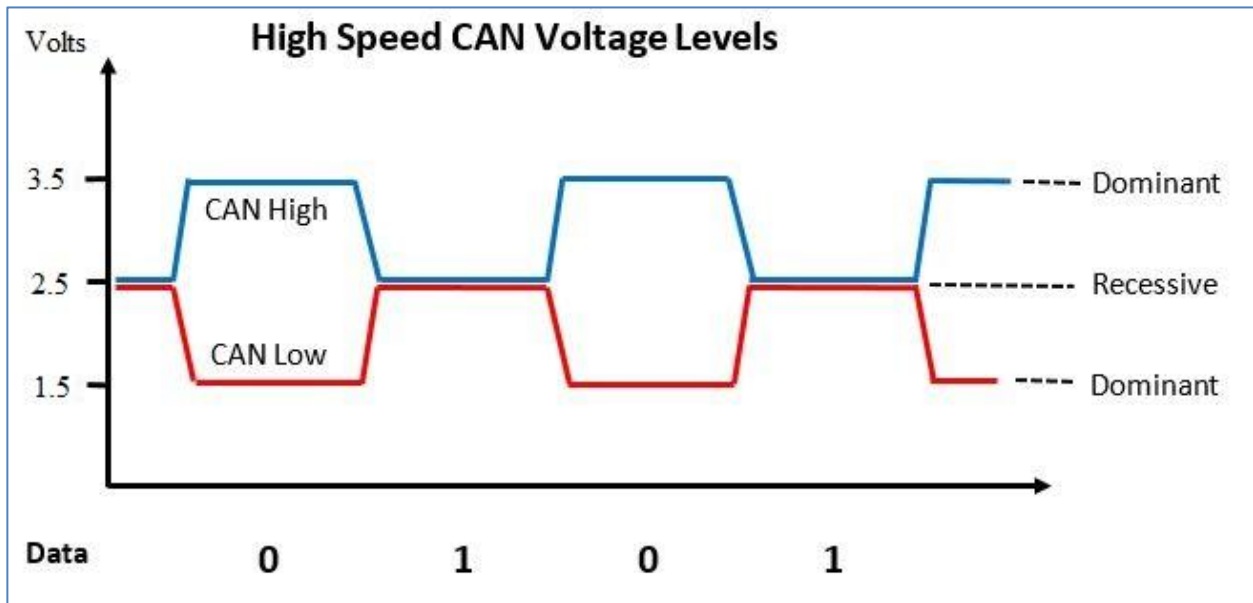


Figura 69. Niveles de tensión en bus CAN

Como se puede observar en la figura existen dos posibles estados en el bus CAN completo que son:

- Dominante: CAN H a 3.5V y el CAN L a 1.5V lo cual se corresponde con un 0 lógico a nivel de mensaje binario.
- Recesivo: CAN H y CAN L a 2.5V lo cual se corresponde con un 1 lógico a nivel de mensaje binario.

Con esta información ya comprendemos como funciona físicamente un bus CAN y por lo tanto podremos comprobar algunos aspectos de nuestro sistema con ayuda de un osciloscopio y un multímetro. Las pruebas que vamos a realizar son:

- Con el sistema sin alimentar comprobar que entre CAN L y CAN H se ve una resistencia de 60Ω.
- Con el sistema alimentado comprobar que en tensión se ve el estado dominante o el estado recesivo.
- Comprobar con el osciloscopio que al enviar un mensaje desde la placa de desarrollo en el bus CAN ocurren cambios.

Realizando estas pruebas podremos estar seguros de que el sistema está bien montado, aunque no podremos saber si la información se está enviando correctamente ya que los mensajes CAN son muy largos (hasta unos 108 bits) y comprobar si el mensaje coincide con un osciloscopio es bastante complicado así que eso lo comprobaremos al montar la segunda parte y ver que llega a la placa LPCXpresso que recibirá los datos.

Realizamos la primera parte del montaje (NUCLEOL4R5Z1P – MCP2551 – Bus CAN) y antes de dar energía a la placa y el micro conectándolos al PC medimos la impedancia en el bus CAN y obtenemos un valor de 63.4Ω, valor bastante cercano a los 60Ω ideales que esperábamos obtener entre CAN H y CAN L y valor que nos es más que suficiente para realizar las pruebas. Este valor es un poco más lejano de 60Ω de lo que debería porque se ha usado como segunda resistencia de 120Ω dos resistencias de 270Ω en paralelo (que equivalen a una de 135Ω) ya que no se disponía de más resistencias de 120Ω, pero como ya se ha visto al asociarse todas en paralelo la diferencia en la resultante bastante baja y podemos realizar las pruebas sin problema.

A continuación, se alimenta el sistema y se mide la tensión en el bus CAN obteniéndose un valor de 2.558V entre CAN H y tierra y entre CAN L y tierra. Esto es buena señal ya que se esperaba obtener un valor de 2.5V



en ambas debido a que el sistema se encuentra en estado recesivo.

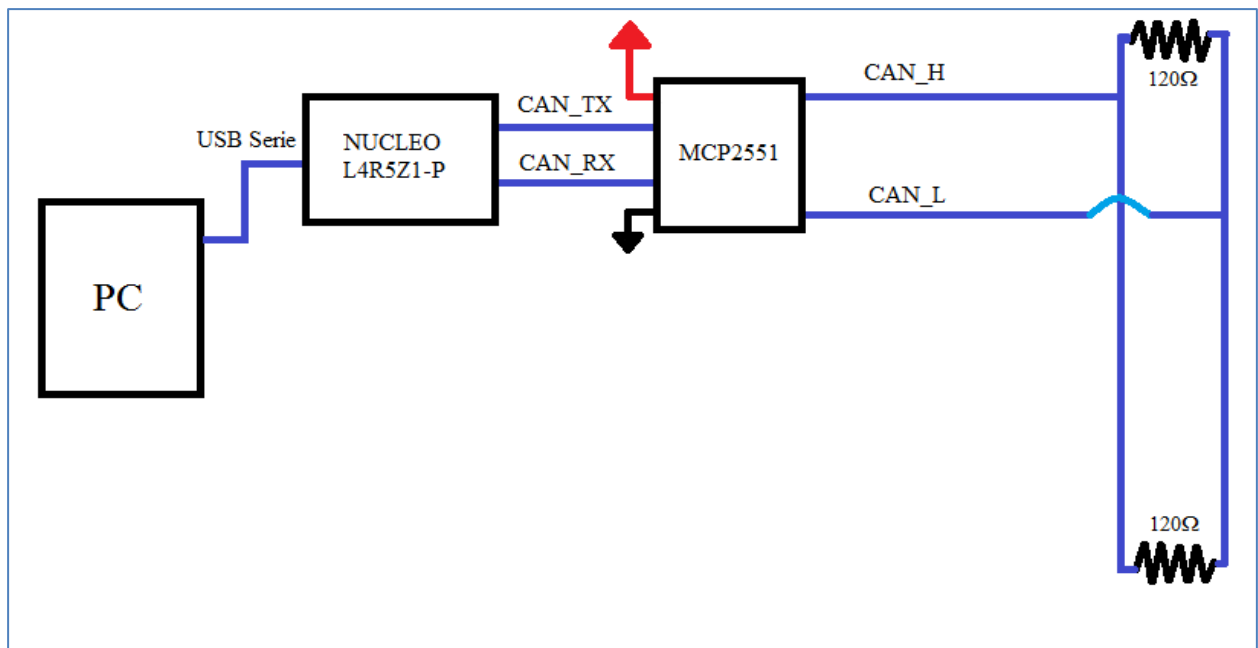


Figura 70. Esquema de la primera parte del montaje para la comunicación CAN

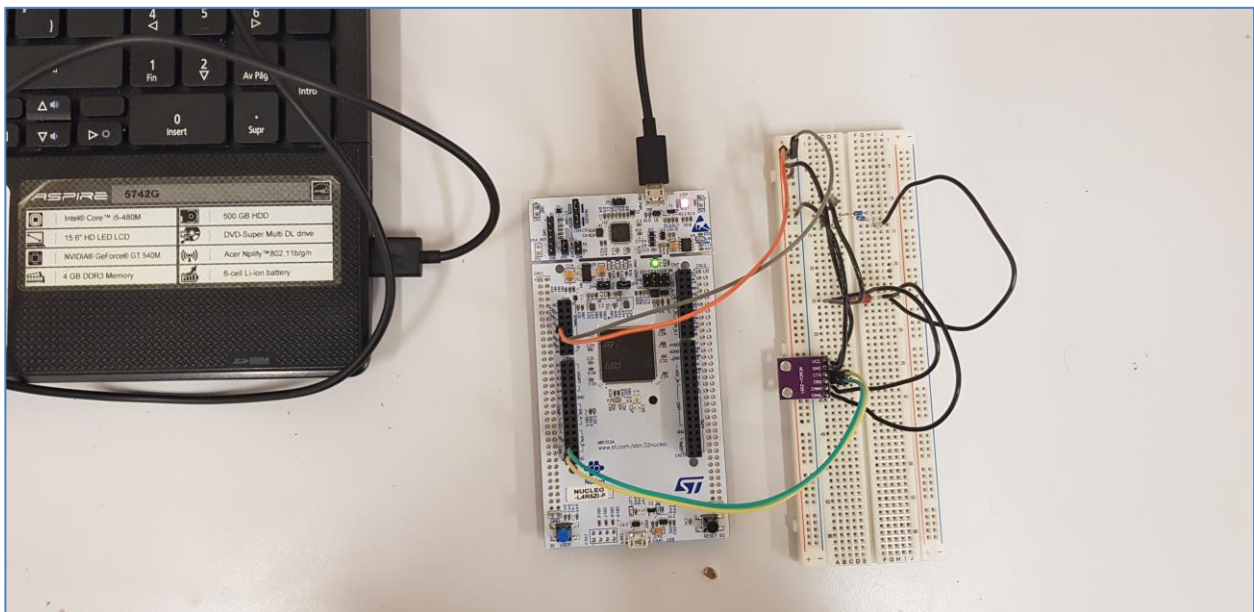


Figura 71. Montaje del circuito de prueba

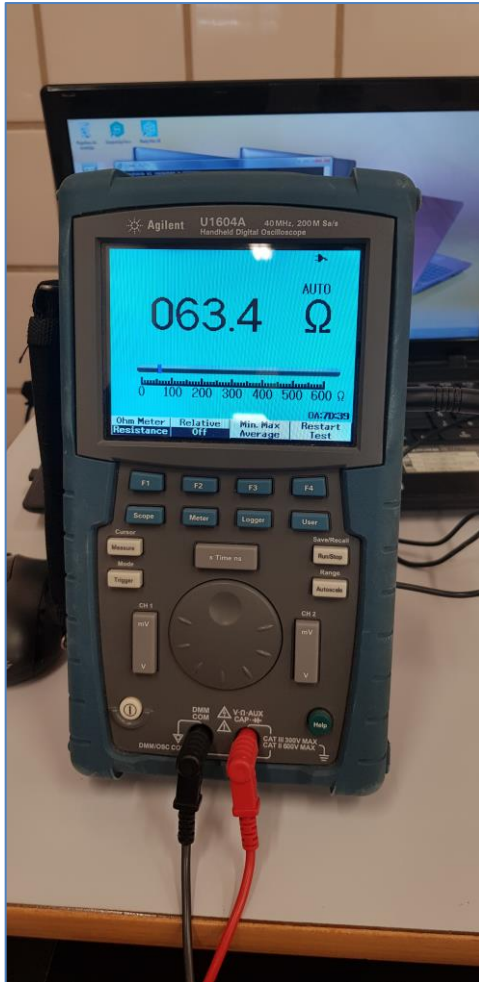


Figura 72. Medida de impedancia

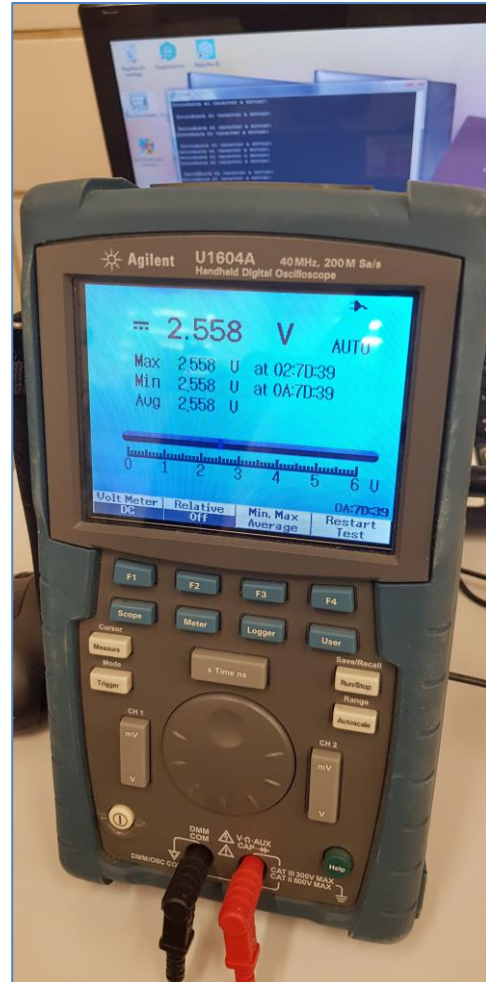


Figura 73. Medida de tensión

Por último, enviamos un mensaje (carácter) desde el PC a través del puerto serie a la placa de desarrollo para que esta lo transmita al bus CAN, esto lo conseguimos a través de un programa cargado en la placa y compilado en Mbed usando su API propia para este tipo de comunicación.

```

1 #include "mbed.h" // Incluimos la libreria principal del RTOS
2
3 CAN can1(PD_0, PD_1); // Definimos un objeto para la comunicación CAN
4 Serial pc(USBTX,USBRX); // Definimos un puerto serie USB
5
6 // Programa principal
7 int main()
8 {
9     char k[8];
10    int i;
11    CANMessage msg; // Definimos un mensaje CAN vacío
12    can1.frequency(9600); // Definimos la frecuencia del bus CAN
13    while(1) {
14        // Se espera a que se introduzca carácter a carácter el mensaje de 8 bytes
15        pc.printf("\r\n Introduzca el mensaje: ");
16        for(i=0; i<8; i++) {
17            k[i]=pc.getc();
18            pc.printf("%c \r\n", k[i]); // Imprimimos por pantalla el carácter introducido
19        }
20
21        msg.id=1400; // Definimos el identificador del mensaje CAN
22
23        //Guardamos k en el mensaje CAN
24        for(i=0; i<8; i++) {
25            msg.data[i]=k[i];
26        }
27        // Imprimimos por pantalla el ID del mensaje CAN
28        pc.printf("Id: %i \r\n",msg.id);
29        // Imprimimos por pantalla los bytes enviados uno a uno
30        pc.printf("B0:%c , B1:%c , B2:%c , B3:%c \r\n B4:%c , B5:%c , B6:%c , B7:%c \r\n \r\n",msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7]);
31        // Mandamos el mensaje CAN por el bus
32        can1.write(msg);
33        // Guardamos los caracteres enviados en una cadena para incluir manualmente el final de cadena y evitar problemas
34        char men[] = {msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7], '\0'};
35        // Imprimimos por pantalla el mensaje total enviado
36        pc.printf("Mensaje: %s \r\n", men);
37    }
38 }
39

```

Figura 74. Código para el envío de mensaje CAN

Y obtenemos la siguiente respuesta en el osciloscopio:

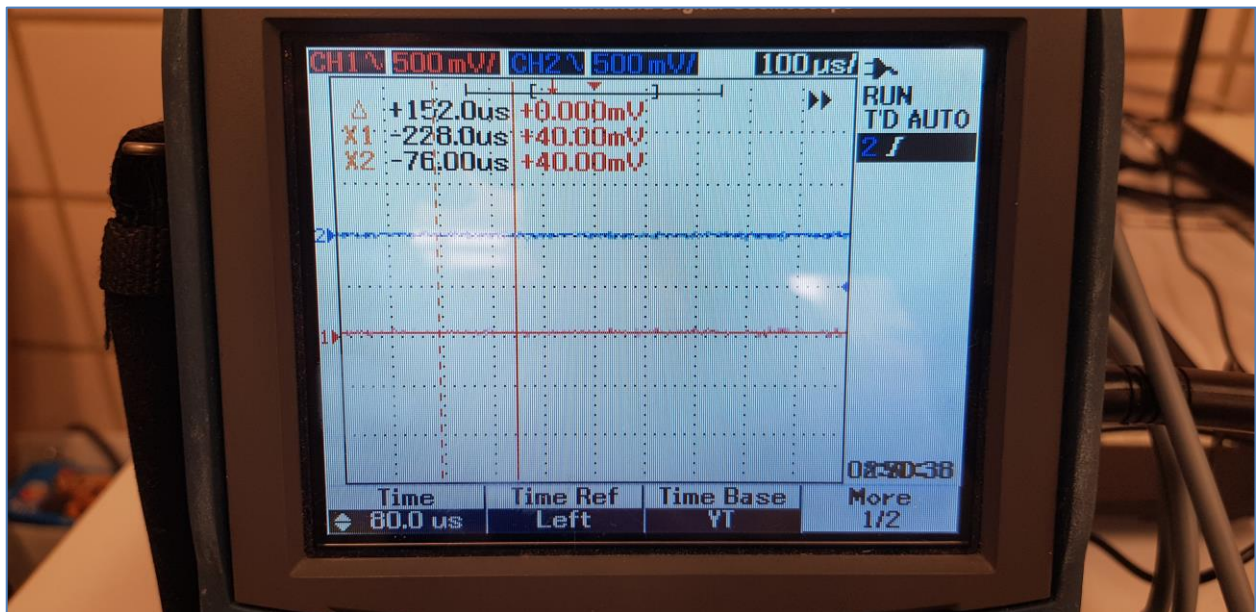


Figura 75. Lectura de CAN H y CAN L mientras no se envía mensaje

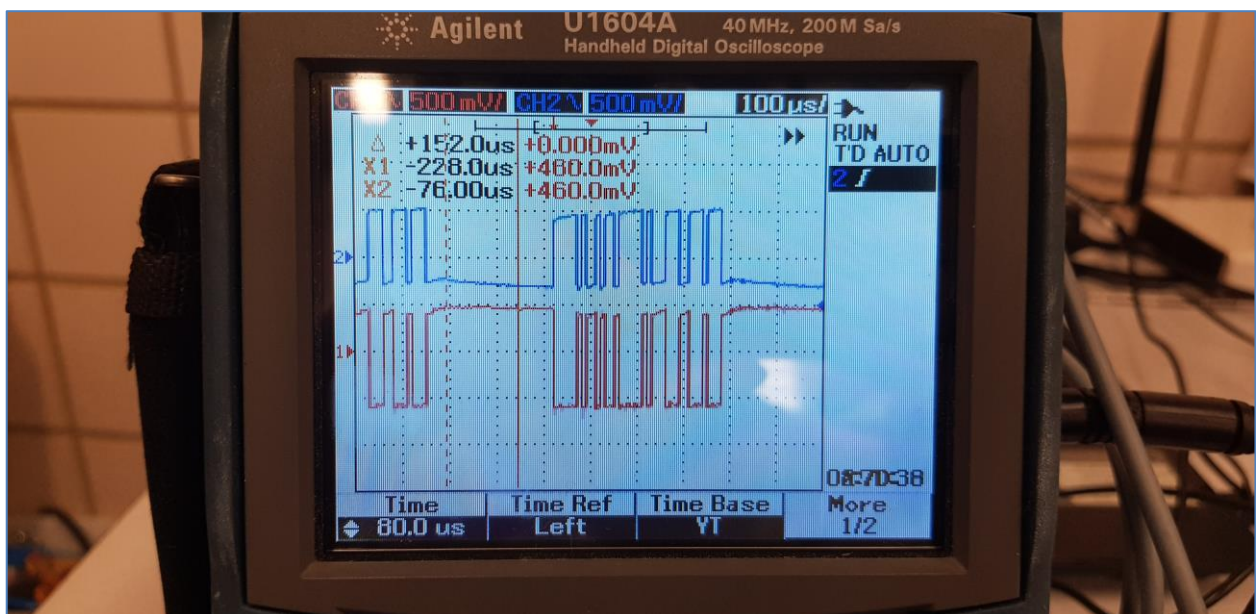


Figura 76. Lectura de CAN H y CAN L mientras se envía un carácter

Como puede observarse en las imágenes la forma de las ondas en el momento del envío del mensaje es la adecuada para una comunicación CAN por lo que podemos decir que se está enviando un mensaje por el bus.

El valor de la tensión de CAN H y CAN L en modo recesivo es de, más o menos, 2.5V en ambas líneas, pero una de las sondas está desplazada en la medida AC para que se distingan mejor.

Además, en las medidas obtenidas del osciloscopio podemos observar que CAN H trabaja entre 2.5V y unos 3.3V, lo que es bastante parecido a lo que esperábamos, y CAN L trabaja entre 2.5V y 1.5V que es lo que esperábamos según la teoría.

Teniendo toda esta información podemos decir que la parte de emisión de nuestro montaje para la comunicación CAN está funcionando correctamente, ahora queda comprobar que la parte de recepción de la información también funcione bien.

Para comprobar el funcionamiento de la parte receptora del sistema de comunicación CAN montaremos el sistema completo y comprobaremos si la placa de desarrollo LPCXpresso54608 recibe algo a través del bus CAN, para ello utilizaremos un programa compilado en Mbed que utilizará una librería disponible para nuestro



Shield de comunicación CAN la cual que nos permitirá controlar el intercambio de información a través de este dispositivo conectado al bus CAN de una manera muy sencilla (parecida a la API proporcionada por Mbed para el control de la comunicación CAN) de esta manera no tendremos que preocuparnos por el paso de la información de CAN a SPI o viceversa ya que de esto se encargará la librería. La librería contiene las clases *SEED\_CAN* y *SEED\_CANMessage*.

### SEED\_CAN Class Reference

```
#include <seed_can.h>
```

#### Public Member Functions

```
SEED_CAN (PinName ncs=SEED_CAN_CS, PinName irq=SEED_CAN_IRQ, PinName mosi=SEED_CAN_MOSI, PinName miso=SEED_CAN_MISO, PinName clk=SEED_CAN_CLK, int spiBitrate=1000000)
Seed Studios CAN-BUS Shield Constructor - Create a SEED_CAN interface connected to the specified pins.
int open (int canBitrate=100000, Mode mode=Normal)
Open initialises the Seed Studios CAN-BUS Shield.
void monitor (bool silent)
Puts or removes the Seed Studios CAN-BUS shield into or from silent monitoring mode.
int mode (Mode mode)
Change the Seed Studios CAN-BUS shield CAN operation mode.
int frequency (int canBitRate)
Set the CAN bus frequency (Bit Rate)
int read (SEED_CANMessage &msg)
Read a CAN bus message from the MCP2515 (if one has been received)
int write (SEED_CANMessage msg)
Write a CAN bus message to the MCP2515 (if there is a free message buffer)
int mask (int maskNum, int canId, CANFormat format=CANStandard)
Configure one of the Acceptance Masks (0 or 1)
int filter (int filterNum, int canId, CANFormat format=CANStandard)
Configure one of the Acceptance Filters (0 through 5)
unsigned char rderror (void)
Returns number of message reception (read) errors to detect read overflow errors.
unsigned char tderror (void)
Returns number of message transmission (write) errors to detect write overflow errors.
int errors (ErrorType type=AnyError)
Check if any type of error has been detected on the CAN bus.
unsigned char errorFlags (void)
Returns the contents of the MCP2515's Error Flag register.
void attach (void(*fptr)(void), IrqType event=RxAny)
Attach a function to call whenever a CAN frame received interrupt is generated.
template<typename T >
void attach (T *tptr, void(T::*mptr)(void), IrqType event=RxAny)
Attach a member function to call whenever a CAN frame received interrupt is generated.
int interrupts (IrqType type)
Check if the specified interrupt event has occurred.
unsigned char interruptFlags (void)
Returns the contents of the MCP2515's Interrupt Flag register.
```

Figura 77. Funciones de la clase *SEED\_CAN*

### SEED\_CANMessage Class Reference

```
#include <seed_can.h>
```

Inherits CAN\_Message.

#### Public Member Functions

```
SEED_CANMessage ()
Creates empty CAN message.
SEED_CANMessage (int _id, const char *_data, char _len=8, CANType _type=CANData, CANFormat _format=CANStandard)
Creates CAN message with specific content.
SEED_CANMessage (int _id, CANFormat _format=CANStandard)
Creates CAN remote message.
```

Figura 78. Funciones de la clase *SEED\_CANMessage*

Para realizar la comunicación CAN completa necesitaremos el código para el envío del mensaje CAN cargado en la placa NUCLEO-L4R5ZI-P y el código para la recepción del mensaje en la placa LPCXpresso54608.

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2 #include "seeed_can.h" // Incluimos la librería para el control del CAN bus shield
3
4 SEED_CANMessage msg; // Definimos un mensaje CAN vacío
5 SEED_CAN spi(P3_30,D9,P3_21,P3_22,P3_20,100000); // Definimos el objeto para el control de la comunicación CAN a través del Shield
6 Serial pc(USBTX,USBRX); // Definimos un puerto serie USB
7 int r;
8
9 // Programa principal
10 int main()
11 {
12     spi.open(9600, SEED_CAN::Normal); // Inicializamos el bus CAN
13     while (1) {
14         // Si recibimos un mensaje a través del bus
15         if(spi.read(msg)) {
16             // Guardamos el ID del mensaje
17             r=msg.id;
18             // Imprimimos el identificador por pantalla
19             pc.printf("Identificador: %d \r\n", r);
20             // Imprimimos por pantalla los bytes recibidos uno a uno
21             pc.printf("B0:%c , B1:%c , B2:%c , B3:%c \r\n B4:%c , B5:%c , B6:%c , B7:%c \r\n",msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7]);
22             // Guardamos los caracteres enviados en una cadena para incluir manualmente el final de cadena y evitar problemas
23             char men[] = {msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7],'\0'};
24             // Imprimimos por pantalla el mensaje recibido completo
25             pc.printf("Mensaje recibido: %s \r\n\r\n", men);
26         }
27     }
28 }
29

```

Figura 79. Código para la recepción del mensaje CAN

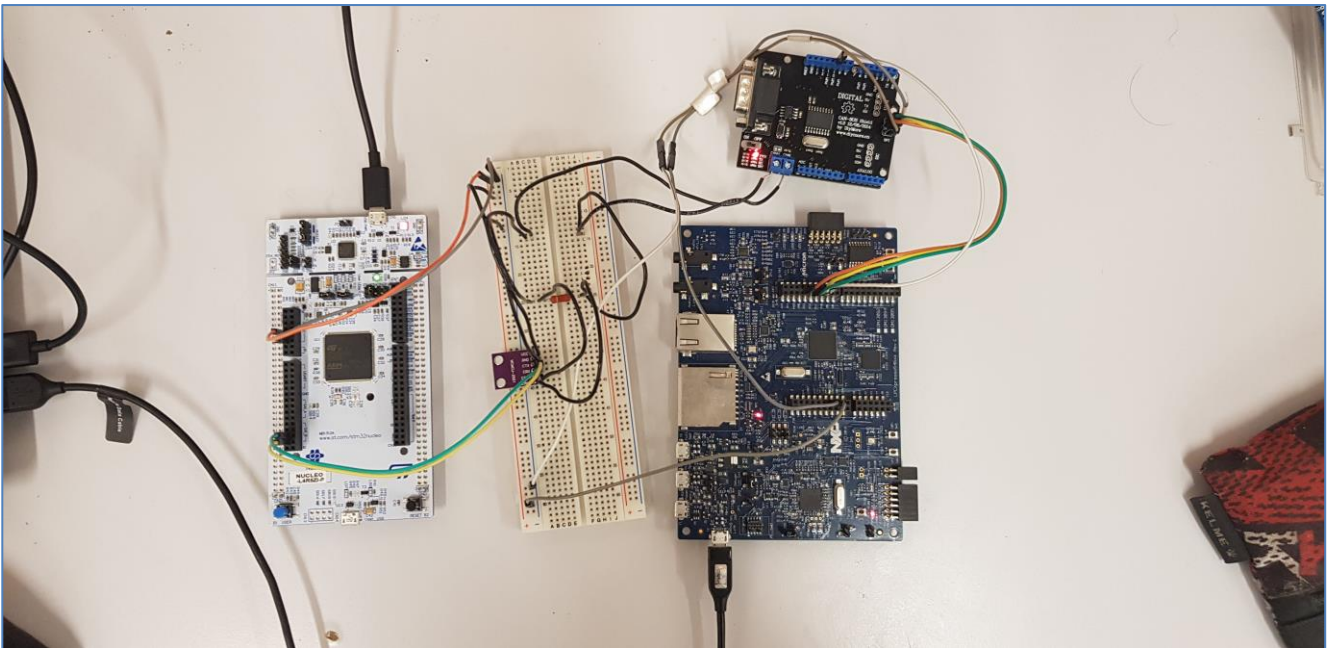


Figura 80. Montaje para comunicación CAN completa

Con el sistema montado y alimentado y ambas placas programadas correctamente abrimos una ventana del Putty para la comunicación puerto serie de cada una de las placas (NUCLEO – COM6 y LPCXpresso – COM4) y realizamos la prueba del envío de información.

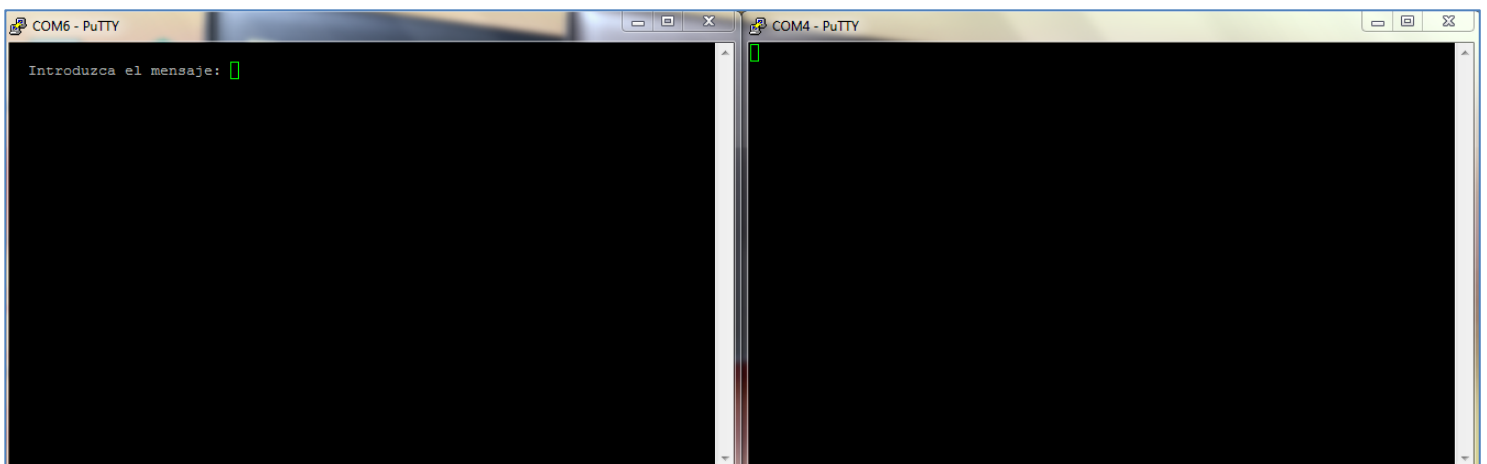
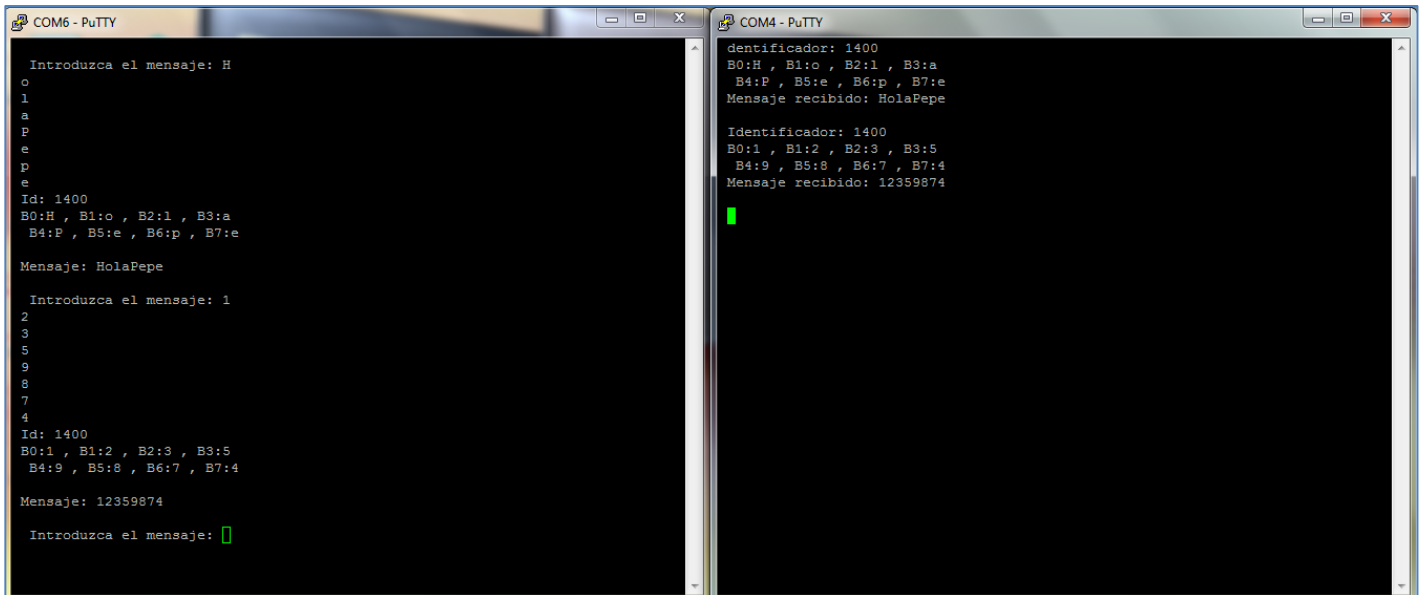


Figura 81. Vista de la información transmitida por el puerto serie de cada una de las placas (Inicio)



```
COM6 - PuTTY
Introduzca el mensaje: H
o
l
a
P
e
P
e
Id: 1400
B0:H , B1:o , B2:l , B3:a
B4:P , B5:e , B6:p , B7:e

Mensaje: HolaPepe

Introduzca el mensaje: 1
2
3
5
9
8
7
4
Id: 1400
B0:1 , B1:2 , B2:3 , B3:5
B4:9 , B5:8 , B6:7 , B7:4

Mensaje: 12359874

Introduzca el mensaje: █

COM4 - PuTTY
Identificador: 1400
B0:H , B1:o , B2:l , B3:a
B4:P , B5:e , B6:p , B7:e
Mensaje recibido: HolaPepe

Identificador: 1400
B0:1 , B1:2 , B2:3 , B3:5
B4:9 , B5:8 , B6:7 , B7:4
Mensaje recibido: 12359874
█
```

Figura 82. Vista de la información transmitida por puerto serie de cada una de las placas (Mensajes enviados)

Como podemos observar en las capturas de la información capturada en el puerto serie de cada placa la transmisión del mensaje CAN se produce de manera totalmente correcta.

A partir de estos programas podríamos realizar modificaciones en el código para, por ejemplo, crear una máscara en la recepción para que solo se lean los mensajes con unos identificadores concretos y poder así conectar al mismo bus CAN más dispositivos sin que se afecten unos a otros si no es lo que se quiere. En este caso no se ha añadido, ya que solo contamos con dos dispositivos conectados al bus y se quiere que se comuniquen.

Con esto se ha conseguido resolver el problema inicial que no nos permitía usar la API de Mbed para la comunicación CAN en la LPCXpresso54608 y finalmente hemos podido controlar esta comunicación de manera sencilla debido a que Mbed contaba con una librería para el Shield CAN que hemos usado, pero esto no ocurrirá siempre y por lo tanto parece que Mbed no es tan compatible entre dispositivos como se pensaba al inicio.

### 4.3 Sistema final

El sistema final que se quería desarrollar como un ejemplo más complejo del uso de la plataforma Mbed consistía en un acelerómetro comunicado por bus CAN con la placa de desarrollo y esta comunicada por medio de ethernet (ya que no contamos con un módulo de conexión inalámbrica) con un servidor TCP, simulando así un sistema de adquisición de vibraciones de un coche. El problema, como se ha visto en el apartado anterior, surge cuando la placa con conexión ethernet que se estaba usando en el proyecto no consigue usar la API de Mbed para el CAN y, por lo tanto, se ha decidido montar un sistema que comunicará las dos placas de las que disponemos por medio de un bus CAN (aunque en el caso de la LPCXpresso54608 se deba controlar este bus mediante SPI) la NUCLEO-L4R5ZI-P enviará información actuando como si fuera el acelerómetro y la LPCXpresso54608 recibirá esta información y la subirá al servidor TCP.

Este montaje final tendría un esquema como este:

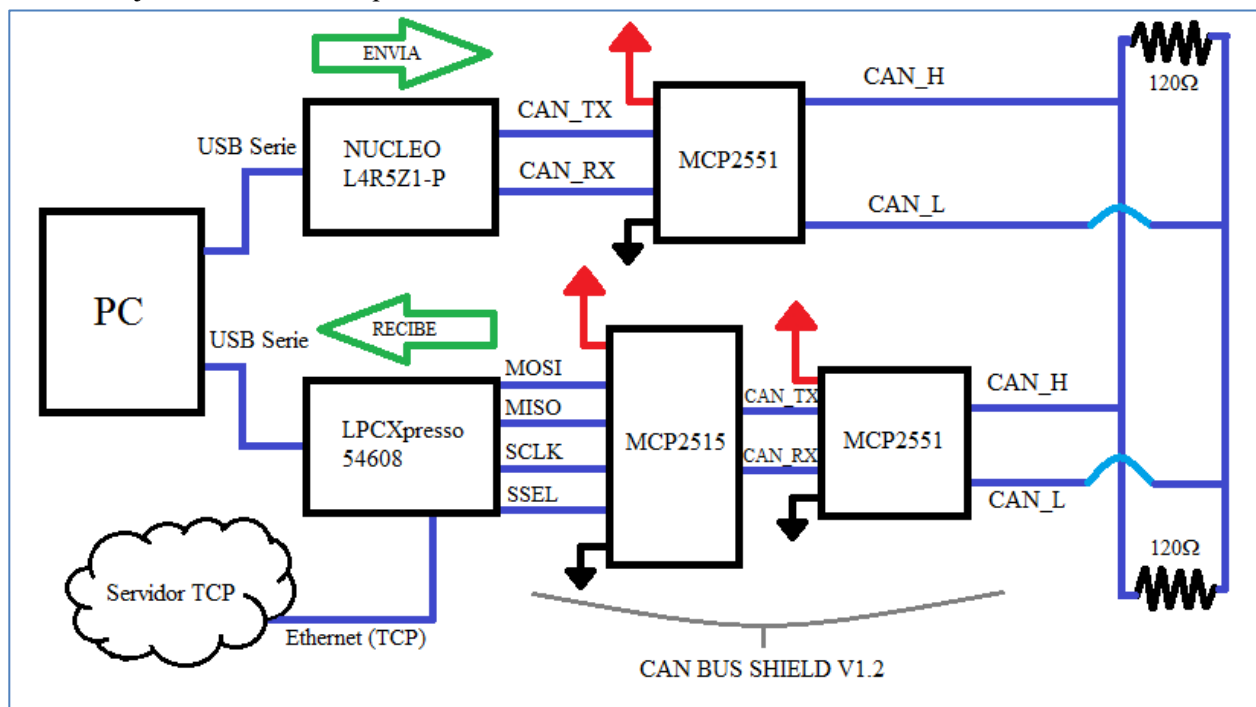


Figura 83. Esquema del sistema final

Siguiendo este esquema, que como se puede observar se basa en el de la comunicación CAN del apartado anterior, montaremos el sistema completo y se realizarán unas pruebas para comprobar que todo funciona correctamente, es decir, que la información llega de la NUCLEO hasta el servidor TCP sin problemas.

Viendo los resultados obtenidos en el apartado de “Comunicación CAN-SPI entre dos dispositivos” podemos suponer que la parte de la transmisión de información por medio del bus CAN funciona, por lo tanto, solo sería necesario incluir en el código de la placa receptora (LPCXpresso54608) las órdenes para que esta información se mande al servidor TCP al ser recibida y comprobar que el servidor recibe la información.

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2
3 CAN can1(PD_0, PD_1); // Definimos un objeto para la comunicación CAN
4 Serial pc(USBTX,USBRX); // Definimos un puerto serie USB
5
6 // Programa principal
7 int main()
8 {
9     char k[8];
10    int i;
11    CANMessage msg; // Definimos un mensaje CAN vacío
12    can1.frequency(9600); // Definimos la frecuencia del bus CAN
13    while(1) {
14        // Se espera a que se introduzca carácter a carácter el mensaje de 8 bytes
15        pc.printf("\r\n Introduzca el mensaje: ");
16        for(i=0; i<8; i++) {
17            k[i]=pc.getc();
18            pc.printf("%c\r\n", k[i]); // Imprimimos por pantalla el carácter introducido
19        }
20
21        msg.id=1400; // Definimos el identificador del mensaje CAN
22
23        //Guardamos k en el mensaje CAN
24        for(i=0; i<8; i++) {
25            msg.data[i]=k[i];
26        }
27        // Imprimimos por pantalla el ID del mensaje CAN
28        pc.printf("Id: %i\r\n",msg.id);
29        // Imprimimos por pantalla los bytes enviados uno a uno
30        pc.printf("B0:%c , B1:%c , B2:%c , B3:%c\r\n B4:%c , B5:%c , B6:%c , B7:%c\r\n\r\n",msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7]);
31        // Mandamos el mensaje CAN por el bus
32        can1.write(msg);
33        // Guardamos los caracteres enviados en una cadena para incluir manualmente el final de cadena y evitar problemas
34        char men[] = {msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7],'\0'};
35        // Imprimimos por pantalla el mensaje total enviado
36        pc.printf("Mensaje: %s\r\n", men);
37    }
38 }
39

```

Figura 84. Código para el envío de mensaje CAN (Cargado en NUCLEO-L4R5ZI-P)

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2 #include "seeed_can.h" // Incluimos la librería para el control del CAN bus shield
3 #include "EthernetInterface.h" // Incluimos la librería para la interfaz ethernet
4
5 SEED_CANMessage msg; // Definimos un mensaje CAN vacío
6 SEED_CAN spi(P3_30,D9,P3_21,P3_22,P3_20,100000); // Definimos el objeto para el control de la comunicación CAN a través del Shield
7 Serial pc(USBTX,USBRX); // Definimos un puerto serie USB
8 EthernetInterface red; // Creamos el objeto para la conexión ethernet
9 int r;
10
11 // Programa principal
12 int main()
13 {
14     spi.open(9600, SEED_CAN::Normal); // Inicializamos el bus CAN
15
16     // Configuramos la interfaz Ethernet y imprimimos por pantalla nuestra dirección IP.
17     printf("\r\n Leyendo dirección ethernet: \r\n");
18     red.connect();
19     const char *ip = red.get_ip_address();
20     printf("La dirección IP del dispositivo es: %s\r\n", ip ? ip : "No IP");
21
22     TCPSocket socket; // Creamos el socket TCP
23
24     while (1) {
25         // Si recibimos un mensaje a través del bus
26         if(spi.read(msg)) {
27             // Guardamos el ID del mensaje
28             r=msg.id;
29             // Imprimimos el identificador por pantalla
30             pc.printf("Identificador: %d\r\n", r);
31             // Imprimimos por pantalla los bytes recibidos uno a uno
32             pc.printf("B0:%c , B1:%c , B2:%c , B3:%c\r\n B4:%c , B5:%c , B6:%c , B7:%c\r\n",msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7]);
33             // Guardamos los caracteres enviados en una cadena para incluir manualmente el final de cadena y evitar problemas
34             char men[] = {msg.data[0], msg.data[1], msg.data[2], msg.data[3], msg.data[4], msg.data[5], msg.data[6], msg.data[7],'\0'};
35             // Imprimimos por pantalla el mensaje recibido completo
36             pc.printf("Mensaje recibido: %s\r\n\r\n", men);
37
38             // Nos conectamos a la red ethernet
39             red.connect();
40             const char *ip = red.get_ip_address();
41
42             //Configuramos el socket TCP
43             TCPSocket socket; // Creamos el socket TCP
44             socket.open(&red);
45             socket.connect("54.229.53.222", 8789);
46
47             //Mandamos la información al servidor
48             socket.send(men,sizeof men);
49         }
50     }
51 }

```

Figura 85. Código para la recepción de mensajes CAN y subida de información a servidor TCP (Cargado en LPCXpresso54608)

Con estos programas cargados en las placas realizamos el montaje del sistema y ya podremos realizar las pruebas.



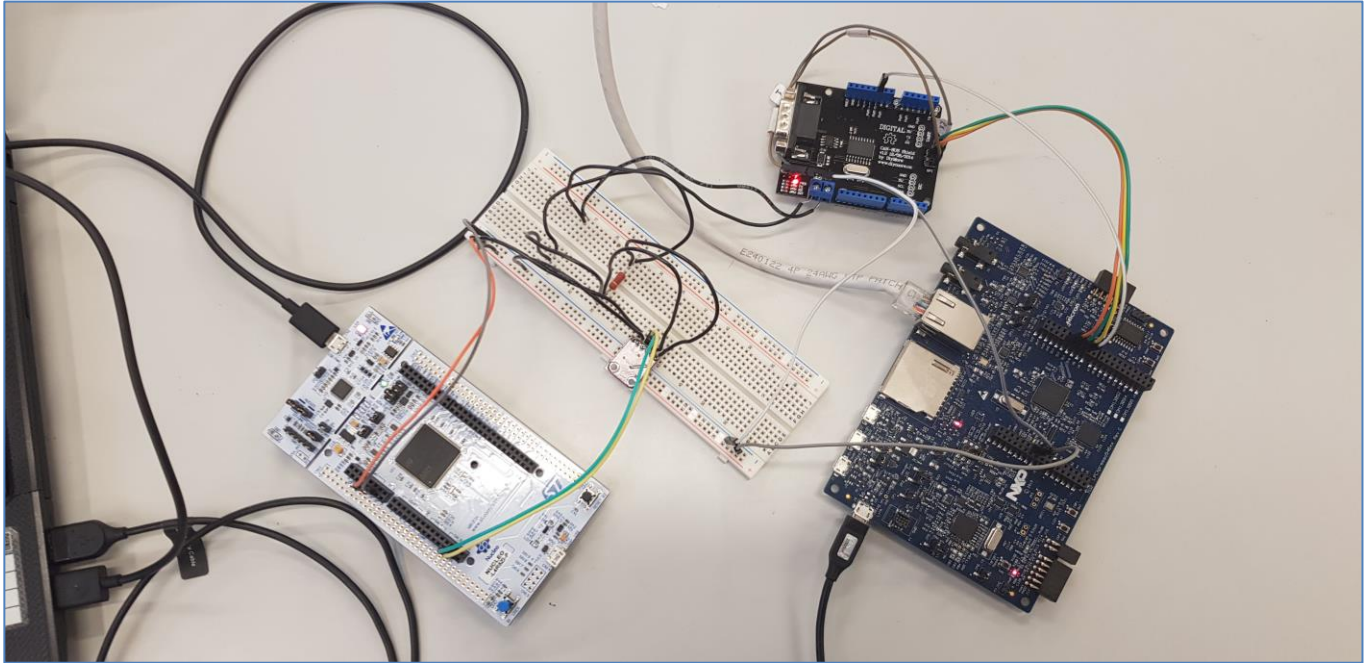


Figura 86. Montaje del sistema final

Observando lo que ocurre a través de los puertos serie de cada placa y en el servidor TCP vemos que:

A screenshot of a PuTTY terminal window titled "COM6 - PuTTY". The window displays the following text:

```
Introduzca el mensaje: H
o
l
a
P
e
p
e
Id: 1400
B0:H , B1:o , B2:l , B3:a
  B4:P , B5:e , B6:p , B7:e

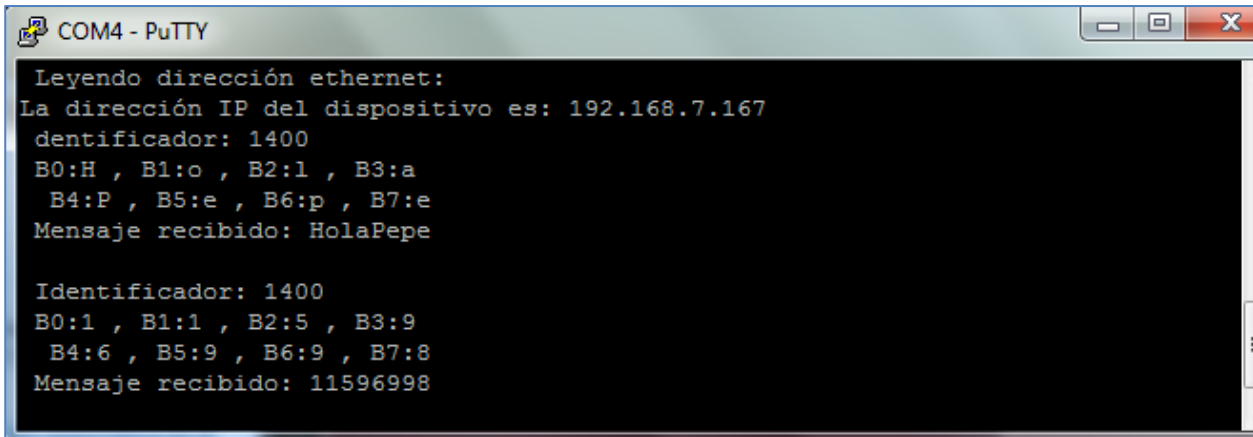
Mensaje: HolaPepe

Introduzca el mensaje: 1
1
5
9
6
9
9
8
Id: 1400
B0:1 , B1:1 , B2:5 , B3:9
  B4:6 , B5:9 , B6:9 , B7:8

Mensaje: 11596998

Introduzca el mensaje: █
```

Figura 87. Vista de la información transmitida por puerto serie de la placa NUCLEO (Envió)



```

COM4 - PuTTY
Leyendo dirección ethernet:
La dirección IP del dispositivo es: 192.168.7.167
identificador: 1400
B0:H , B1:o , B2:l , B3:a
  B4:P , B5:e , B6:p , B7:e
Mensaje recibido: HolaPepe

Identificador: 1400
B0:1 , B1:1 , B2:5 , B3:9
  B4:6 , B5:9 , B6:9 , B7:8
Mensaje recibido: 11596998

```

Figura 88. Vista de la información transmitida por puerto serie de la placa LPCXpresso (Recepción y subida a servidor)



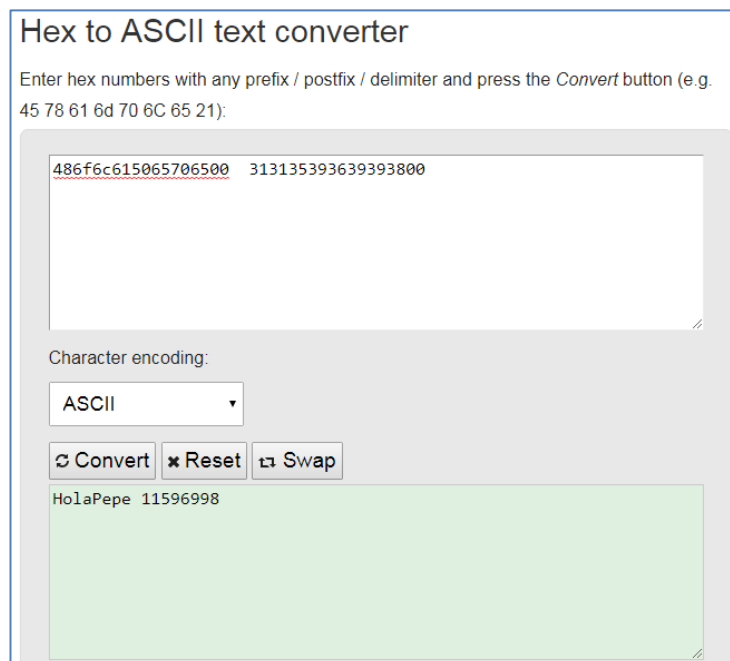
```

{
  "data": "486f6c615065706500",
  "id": "34",
  "timestamp": "2019-06-28 10:53:19"
},
{
  "data": "313135393639393800",
  "id": "35",
  "timestamp": "2019-06-28 10:53:53"
}

```

Figura 89. Mensajes recibidos por el servidor

Como podemos observar en las imágenes, la transmisión de información a través del bus CAN sigue funcionando sin problema y además el servidor recibe los mensajes que se han enviado. Para confirmar que los mensajes que se han recibido en el servidor son los correctos utilizamos un convertidor HEX-ASCII que nos confirma que los mensajes han llegado correctamente.



Hex to ASCII text converter

Enter hex numbers with any prefix / postfix / delimiter and press the *Convert* button (e.g. 45 78 61 6d 70 6C 65 21):

486f6c615065706500 313135393639393800

Character encoding: ASCII

Convert Reset Swap

HoLaPepe 11596998

Figura 90. Conversión de hexadecimal a texto ASCII de los mensajes recibidos en el servidor

Con esto se ha conseguido emular el acelerómetro con conexión CAN por medio de la NUCLEO-L4R5ZI-P y mediante la LPCXpresso54608 se ha subido esta información a un servidor TCP, por lo que se ha realizado la simulación de un dispositivo IoT con sensor de vibraciones en un automóvil correctamente.

## 4.4 Compatibilidad de las aplicaciones desarrolladas

Una de las ventajas que señalamos del uso de Mbed como RTOS y entorno de desarrollo era que un código desarrollado para un dispositivo podía ser utilizado en otro con las funcionalidades necesarias simplemente cambiando los pines usados por los correspondientes en el nuevo dispositivo. Esto era una ventaja bastante grande del RTOS, pero como se ha visto con anterioridad no todos los dispositivos aceptan todas las APIs de Mbed y por lo tanto esta compatibilidad no será total, por ejemplo, una aplicación que use comunicación CAN en la NUCLEO-L4R5ZI-P no funcionará en la placa de desarrollo LPCXpresso54608 ya que, esta no soporta la API para CAN de Mbed.

Teniendo esto en cuenta se va a comprobar la compatibilidad de algunas de las aplicaciones desarrolladas para la placa LPCXpresso54608 implementándolas en la NUCLEO-L4R5ZI-P, el código no se tocará más que para modificar los pines usados.

### 4.4.1 Prueba de entorno de programación

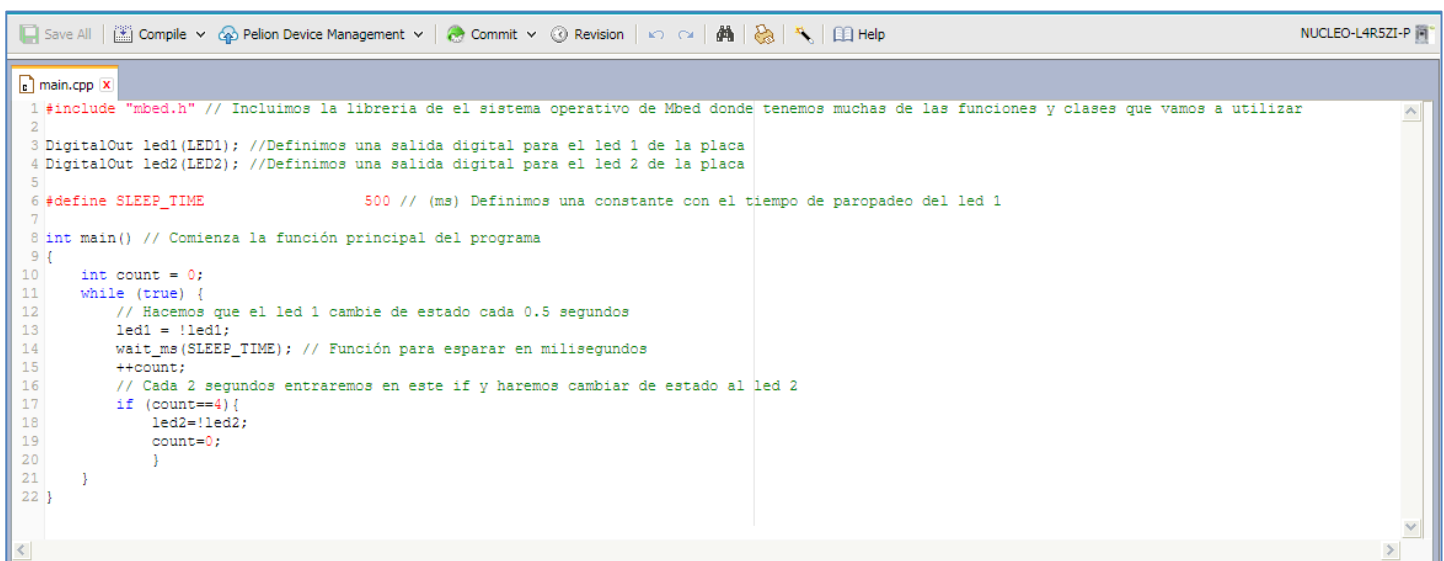
Este es el programa de ejemplo que se usó para la placa de desarrollo LPCXpresso54608 con el que se hacía que un led cambiara de estado cada 0.5 segundos y otro cada dos segundos.

Ahora vamos a implementar el mismo código en la placa de desarrollo NUCLEO-L4R5ZI-P, además en este caso será exactamente el mismo código ya que los pines utilizados son los de los leds y Mbed tiene unos accesos rápidos a ellos en todas las placas con LED1, LED2... por lo tanto ni siquiera habrá que cambiar los pines, bastará con cambiar el dispositivo objetivo en la esquina superior derecha del compilador online.

Al compilar el código, como de costumbre, obtenemos un archivo en binario, lo arrastramos al almacenamiento extraíble que es la placa y está se programa con su contenido.

El resultado parece ser bueno ya que el led 1 (en esta placa verde) se enciende y se apaga cada 0.5 segundos y el led 2 (azul en esta placa) lo hace cada dos segundos.

Se puede decir que en este caso a compatibilidad es total ya que no ha sido necesario tocar nada del código.



```
1 #include "mbed.h" // Incluimos la librería de el sistema operativo de Mbed donde tenemos muchas de las funciones y clases que vamos a utilizar
2
3 DigitalOut led1(LED1); //Definimos una salida digital para el led 1 de la placa
4 DigitalOut led2(LED2); //Definimos una salida digital para el led 2 de la placa
5
6 #define SLEEP_TIME          500 // (ms) Definimos una constante con el tiempo de parpadeo del led 1
7
8 int main() // Comienza la función principal del programa
9 {
10     int count = 0;
11     while (true) {
12         // Hacemos que el led 1 cambie de estado cada 0.5 segundos
13         led1 = !led1;
14         wait_ms(SLEEP_TIME); // Función para esperar en milisegundos
15         ++count;
16         // Cada 2 segundos entraremos en este if y haremos cambiar de estado al led 2
17         if (count==4){
18             led2=!led2;
19             count=0;
20         }
21     }
22 }
```

Figura 91. Código de PruebaEntorno para NUCLEO-L4R5ZI-P

### 4.4.2 Comunicación puerto serie

Aquí vamos a comprobar la compatibilidad de la aplicación desarrollada para leer y escribir a través del puerto serie USB, en este caso el código también será el mismo que el usado para la LPCXpresso54608 ya que Mbed

cuenta con atajos para los puertos serie de USB en todas las placas (USBTX y USBRX).

El único cambio que realizaremos será el cambio de placa por la NUCLEO-L4R5ZI-P en la esquina superior derecha del compilador y obtendremos los siguientes resultados al programar la placa de desarrollo con el binario obtenido.

```

1  #include "mbed.h" // Incluimos la librería principal del RTOS
2
3  DigitalOut myled(LED1); // Salida digital para el led1
4  DigitalOut myled2(LED2); // Salida digital para el led2
5
6  Serial pc(USBTX, USBRX); // Configuración del puerto serie (Puertos TX y RX) para el uso de un USB
7
8  int main() // Inicio de la función principal del programa
9  {
10     char k ;
11     pc.printf(" Introduce un caracter: ");
12     myled = 0; // Enciende el led 1
13     myled2 = 0; // Enciende el led 2
14     //Comienza bucle infinito
15     while(1) {
16         k = pc.getc(); // Se espera a que se introduzca un carácter y se guarda en k
17         pc.printf("%c \r \n", k); // Imprimimos por pantalla el carácter introducido
18         switch(k){
19             case 'a': //Si es una 'a' cambiamos el estado del led 1
20                 myled = !myled;
21                 break;
22
23             case 'b': //Si es una 'b' cambiamos el estado del led 2
24                 myled2 = !myled2;
25                 break;
26
27             case 'c': // Si es una 'c' imprimimos por pantalla: "Hola mundo"
28                 pc.printf("Hola mundo \r \n");
29                 break;
30
31             case 'd': //Si es una 'd' imprimimos por pantalla "Adios"
32                 pc.printf("Adios \r \n");
33                 break;
34
35             default: // En el resto de los casos se imprime "Caracter incorrecto"
36                 pc.printf("Caracter incorrecto \r \n");
37                 break;
38         }
39         pc.printf("Introduce un caracter: ");
40     }
41 }

```

Figura 92. Código para comunicación por puerto serie en NUCLEO-L4R5ZI-P

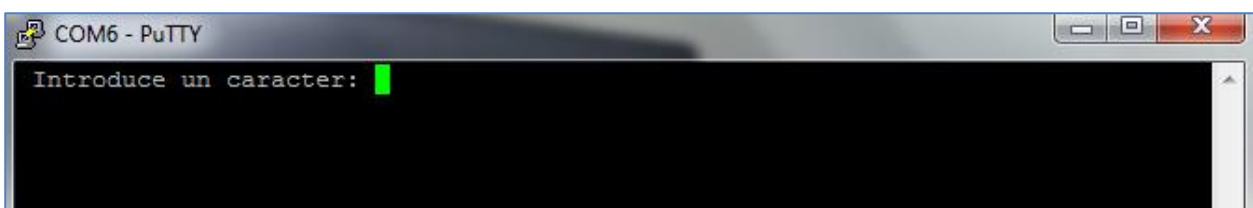
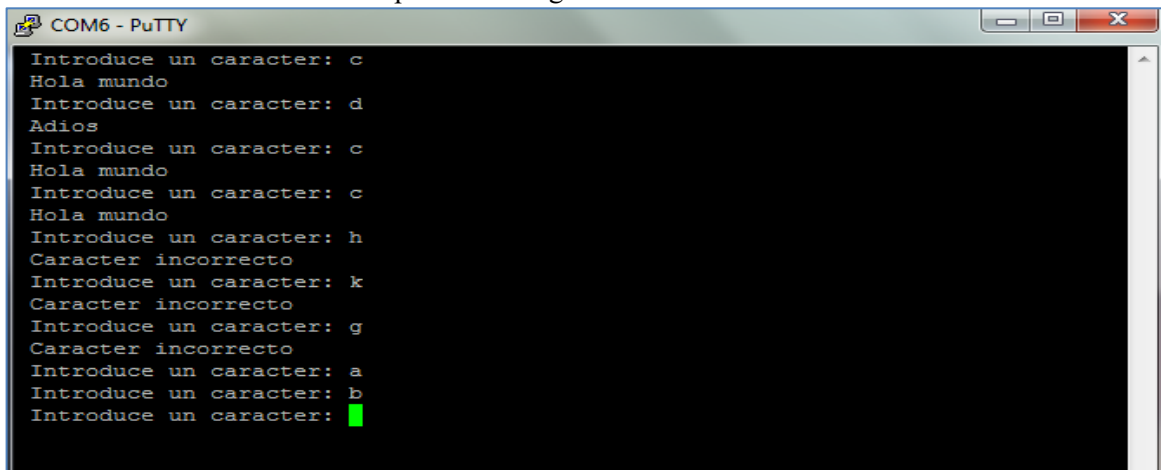


Figura 93. Programa ejecutado y a la espera de recibir algo por el puerto serie

Al introducir una 'a' o una 'b' se observa cómo se encienden o apagan los leds verde (LED1) y azul (LED2) respectivamente como era de esperar que ocurriera.

Y al introducir otros caracteres la respuesta es la siguiente:



```
COM6 - PuTTY
Introduce un caracter: c
Hola mundo
Introduce un caracter: d
Adios
Introduce un caracter: c
Hola mundo
Introduce un caracter: c
Hola mundo
Introduce un caracter: h
Caracter incorrecto
Introduce un caracter: k
Caracter incorrecto
Introduce un caracter: g
Caracter incorrecto
Introduce un caracter: a
Introduce un caracter: b
Introduce un caracter: █
```

Figura 94. Respuesta del programa ante los caracteres introducidos

Comprobamos que el funcionamiento de la aplicación es completamente idéntico al que tenía en la placa LPCXpresso54608 y de nuevo ni siquiera se han tenido que cambiar los pines gracias a unos atajos de Mbed.

Si se usara la comunicación serie para puertos serie no USB bastaría con cambiar los pines TX y RX por los correspondientes a la placa y todo funcionaría igual.

#### 4.4.3 Temporizadores y “tickers”

Por último, vamos a comprobar la compatibilidad de la función desarrollada para controlar los temporizadores del micro de la placa LPCXpresso54608.

Para ello de nuevo usaremos el mismo código de base, pero esta vez cambiaremos el pin usado para el interruptor de SW3 a PC\_13 que es el pin correspondiente al botón de la NUCLEO-L4R5ZI-P.

Los pines de los leds son LED1, LED2 y LED3 que siguen siendo válidos ya que son atajos de Mbed para los leds en todas las placas compatibles. Además, cambiaremos la placa objetivo a la NUCLEO-L4R5ZI-P como en los ejemplos anteriores.

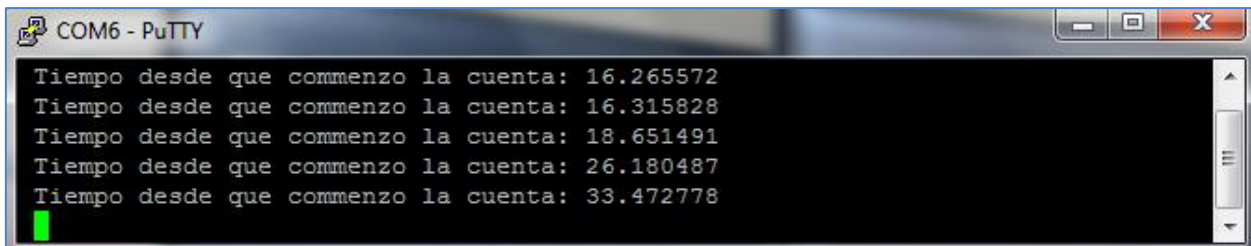
Una vez compilamos, obtenemos el binario correspondiente y lo cargamos en la placa de desarrollo y observamos como el led verde (LED1) comienza a apagarse y encenderse cada segundo y el led azul (LED2) lo hace cada dos segundos. El led rojo (LED3) se mantiene apagado, pero cuando pulsamos el interruptor de usuario este se enciende (o se apaga si ya estaba encendido) y además obtenemos por puerto serie una lectura del tiempo transcurrido desde que se inició la aplicación.

```

1 #include "mbed.h" // Incluimos la librería principal del RTOS
2
3 DigitalOut led1(LED1); // Salida digital para el led1
4 DigitalOut led2(LED2); // Salida digital para el led2
5 DigitalOut led3(LED3); // Salida digital para el led3
6
7 InterruptIn Interruptor(PC_13); // Asignamos una interrupción a un pin, concretamente al interruptor 3 de la placa
8
9 Timer timer; // Creamos el temporizador
10 Ticker ticker1, ticker2; // Creamos dos tickers para dos interrupciones recurrentes
11
12 // Función 1: Cambio de estado del led 1
13 void Pled1() {
14     led1=!led1;
15 }
16 // Función 2: Cambio de estado del led 2
17 void Pled2() {
18     led2=!led2;
19 }
20 // Función 3: Cambio de estado del led 3 y impresión en pantalla del tiempo pasado desde que se inicio el temporizador
21 void Pled3() {
22     led3=!led3;
23     printf("Tiempo desde que comenzo la cuenta: %f \r \n",timer.read());
24 }
25
26 // Función principal
27 int main()
28 {
29     timer.start(); // Inicialización del timer
30     Interruptor.fall(&Pled3); //Llamada de la función 3 cuando se detecte un flanco de bajada den Interruptor (interruptor 3 de la placa)
31     ticker1.attach(&Pled1,1); //Llamada de la función 1 cada segundo
32     ticker2.attach(&Pled2,2); //Llamada de la funcion 2 cada 2 segundos
33
34 }

```

Figura 95. Código del programa de prueba para las clases *Timer* y *Ticker* en la NUCLEO-L4R5ZI-P



```

COM6 - PuTTY
Tiempo desde que comenzo la cuenta: 16.265572
Tiempo desde que comenzo la cuenta: 16.315828
Tiempo desde que comenzo la cuenta: 18.651491
Tiempo desde que comenzo la cuenta: 26.180487
Tiempo desde que comenzo la cuenta: 33.472778

```

Figura 96. Respuesta de la aplicación por puerto serie al pulsar el botón de usuario

Se puede observar que el funcionamiento es igual que en la placa para la cual había sido redactado el código y solo ha sido necesario el cambio de una asignación de un pin.

# 5 CONCLUSIONES

---

Después de esta investigación del uso de Mbed como sistema operativo y entorno de programación para aplicaciones IoT, en la que se han explorado bastantes de sus funciones en distintos dispositivos, se han podido obtener unas conclusiones sobre el uso de este RTOS.

En primer lugar y lo más importante probablemente, se ha descubierto que Mbed no provee de total funcionalidad a placas que, según su web, deberían de ser compatibles. Esto es una de las puntos en contra que se han descubierto durante el trabajo ya que sí, ganaríamos la comodidad que nos ofrece Mbed en algunas aplicaciones que podamos realizar con las funcionalidades que si soporta nuestra tarjeta (véase el caso del acelerómetro con envío de datos al servidor TCP en la LPCXpresso54608) pero para otras aplicaciones perderíamos totalmente esta comodidad, ya que no contaríamos con las APIs necesarias, y además es posible que perdiéramos otras funcionalidades que si tendríamos disponibles en otros entornos de programación, véase como ejemplo la comunicación CAN en la LPCXpresso54608 donde al no poder usar la API para CAN de Mbed perdíamos además la oportunidad de usar los puertos CANTX y CANRX de la placa, teniendo que realizar la comunicación CAN por SPI primero y añadiendo así otro dispositivo al montaje.

Es verdad que en las placas que no disponen de todas las funcionalidades que deberían, se pueden tomar otros caminos para llevar adelante alguna de las aplicaciones, como se ha hecho en este proyecto con la comunicación CAN en la LPCXpresso54608, pero en esos casos se debe recurrir a foros o librerías de terceros (como la que se ha usado) y si no se cuenta ni siquiera con esto se deberá intentar controlar esa funcionalidad de alguna manera a través de otra de las APIs disponibles, perdiendo así toda la ventaja que se supone nos proporcionaba Mbed en este campo.

Uno de los puntos positivos que se ha descubierto durante este trabajo es la sencillez de uso que provee Mbed en las funcionalidades que soporta, pudiendo, por ejemplo, realizar una conexión a un servidor TCP con prácticamente un par de líneas de código o controlar la comunicación CAN de un dispositivo con comandos a tan alto nivel. Además de proveernos de documentación e incluso ejemplos de uso para cada funcionalidad disponible haciendo el uso de estas aún más sencillo.

También, gracias a la realización del trabajo y la búsqueda de información, se puede decir que Mbed da total funcionalidad a los dispositivos con microprocesador de la familia STM32 como, por ejemplo, la placa NUCLEO-L4R5ZI-P que se ha usado durante este trabajo.

Por lo tanto, como conclusión se puede afirmar que Mbed es una herramienta increíblemente útil y cómoda tanto como RTOS como de entorno de programación, pero solo se puede estar seguro de su total compatibilidad con el dispositivo usado si este cuenta con un microprocesador de la familia STM32, sino es posible que algunas de las funcionalidades no estén disponibles y por lo tanto no se pueda sacar todo el partido al dispositivo, lo cual no es recomendable.





# GLOSARIO

---

RTOS: Real Time Operating System ó Sistema operativo de tiempo real

IoT: Internet of Things ó Internet de las cosas

CAN: Controller Area Network (Protocolo de comunicaciones)

ARM: Tipo de arquitectura computacional

C++: Lenguaje de programación de alto nivel

MCU: Microcontroller Unit ó Microcontrolador

USB: Universal Serial Bus (Bus de comunicaciones)

UART: Universal Asynchronous Receiver-Transmitter ó Transmisor-Receptor Asíncrono Universal

I2C: Inter-Integrated Circuit ó Circuito inter-integrado (Bus de comunicaciones)

SPI: Serial Peripheral Interface ó Interfaz periférica en serie (Protocolo de comunicaciones)

MISO: Master Input Slave Output (Señal de SPI)

MOSI: Master Output Slave Input (Señal de SPI)

CS: Select (Señal de SPI)

SCK: Clock (Señal de SPI)

ISP: Internet Service Provider ó Proveedor de servicios de Internet

Pinout: Esquema con la disposición de los terminales de un dispositivo electrónico

Layout: Terminio que significa diseño, plan o disposición y es muy utilizado en el ámbito tecnológico

Ethernet: Protocolo de redes de area local

UDP: User Datagram Protocol (Protocolo de la capa de transporte de Internet)

TCP: Transmission Control Protocol (Protocolo de la capa de transporte de Internet)

API: Application Programming Interface ó Interfaz de programación de aplicaciones

IDE: Integrated Development Environment ó Entorno de desarrollo integrado

Putty: Programa informatico para el control de los puertos serie en un PC