Universidad de La Laguna

ESCUELA SUPERIOR DE INGENIERÍA Y TECNOLOGÍA

TRABAJO DE FIN DE GRADO

REALIZACIÓN DE UN SISTEMA DISTRIBUIDO DE CONTROL INDUSTRIAL UTILIZANDO BUS CAN

Autora: Itziar Rodríguez Hernández

Tutor: Alberto F. Hamilton Castro

Julio de 2018

Índice

Índice de figuras

Índice de tablas

Abreviaturas y siglas

- **DCS** → Distributed Control Systems / Sistema de Control Distribuido.
- **PLC** → Programmable Logical Controller / Controlador Lógico Programable.
- **CAN** → Controller Area Network.
- **ESA** → European Space Agency / Agencia Espacial Europe.
- **DCS** → Distributed Control System / Sistema de Control Distribuido.
- **ISO** → International Organization for Standarization / Organización Internacional de Normalización.
- **BBB** → BeagleBone Black.
- **ARM** → Architecting a Smarter World.
- **PC** → Personal Computer.
- **PID** → Proportional – Integral- Derivative controller / Controlador Proporcional - Integral – Derivativo.
- **PWM** → Pulse Width-Modulation / Modulación por Ancho de Pulsos.
- **CSMA** → Carrier Sense Multiple Access.
- **CD** → Collision Detection.
- **AMP** → Arbitration on Message Priority.
- **ECU** → Electronic Control Unit.
- **ADC** → Analog-to-Digital Convertion / Conversión Analógico-Digital.
- **DAC** → Digital-to-Analog Convertion / Conversión Digital-Analógica.
- **LLC** → Logical Link Control / Subcapa de Control de Enlace Lógico.
- **MAC** → Media Acces Control / Subcapa de Control de Acceso al Medio.
- **PHS** → Physucal Coding Sub- layer / Subcapa de Señalización Física.
- **PMA** → Physical Media Attachement.
- **NRZ** → Non-Return to Zero.
- **SOF** → Start-Of-Frame / Inicio de Trama.
- **RTR** → Remote Transmit Request / Petición de Transmisión Remota.
- **IDE** → Identifier Extension Bit / Bit de Extensión de Identificador.
- **DLC** → Data Length Code / Código de Longitud de Datos.
- **CRC** → Cyclic Redundance Check / Verificación por Redundancia Cíclica.
- **ACK** → Acknowledge slot / Campo de Reconocimiento.
- **EOF** → End-Of-Frame / Fin de Trama.
- **IFS** → Interframe Space / Espacio entre Tramas.
- **CPC_PCI** → Can PC_Peripheral Component Interconnect.
- **USB** → Universal Serial Bus.
- **PCB** → Printed Circuit Board / Placa de Circuitos Impreso.
- **SAR** → Successive Approximation Register / Registro de Aproximación.

Abstract

Nowadays, distributed control systems (DCS) are used in many industrial procedures.

This is consequence of the big advantages that it offers if we compare them with other control systems. DCS allow us to take control of big plants without needing many wires, as well as the Programmable Logical Controller implementation (PLC) in an only bus.

Because of the great advantages they bring us and the importance of its domain in the industrial sector it is why we are carrying out this project.

To develop the distributed control system, it has been combined the use of CAN bus and the BeagleBone Black card, being CAN's protocol one of the most important communication protocols nowadays, and the BeagleBone Black card, one of the most commercialized card.

With the purpose of being in touch with each one of them, we have worked firstly separately to study their characteristics and how they work out. Finally, it has been carried out a distributed control system with the use of both of them to control the temperature of a circuit within a PID controller.

In the elaboration of the project, I have learnt how to use de github (a web- based git [29] repository hosting service), besides being able to programming and dominate CAN bus and the BeagleBone Black card.

Finally, the result of the project has turned out in being able to control the CAN communication and the PWM functions, as well as the analogical- digital conversion of the BeagleBone black card to elaborate a community of programs in C++ language, that worked out with both devices separately and then with both together.

Capítulo 1. Introducción

1.1. Perspectiva general

El protocolo de comunicación CAN (Controller Area Network) surge en el año 1983 de la mano de Bosch.

Comienza a utilizarse de manera masiva en la industria del automóvil una década después de su creación, suponiendo que dicha industria represente el 80% de su uso total.

Sin embargo, en estos últimos años, han incrementado los sectores que precisan de este protocolo para la interconexión de sistemas y/o subsistemas.

En general, podríamos comentar que el bus CAN es ideal para aplicaciones que precisan de un control distribuido en tiempo real en redes sin un gran flujo de datos. Esto se debe a que la topología multimaestro y multidireccional de bus de dos cables CAN permite agregar con facilidad funcionalidad adicional a un sistema, a la vez que, reduce la cantidad de cableado empleado, además, presenta una gran robustez ante la detección y corrección de errores, la tensión superficial y la prioridad no destructiva.

Por todos estos motivos, el protocolo CAN se está abriendo paso de manera importante en diferentes industrias como pueden ser las de transporte público, ascensores, control industrial, automatización de edificios y en el sector espacial, dónde incluso la Agencia Espacial Europea (ESA) está trabajando para reemplazar la arquitectura tradicional de bus de sistemas espaciales por este nuevo protocolo.

La importancia y uso creciente hacen que el dominio y control del protocolo CAN sea una opción de futuro y una apuesta segura en el ámbito de los protocolos de comunicación industrial.

Todo esto sumado a la necesidad que tienen los procesos industriales de estar controlados hace que la combinación entre un proceso de control y el bus CAN sea una pareja idónea.

Para ello, los sistemas de control distribuidos (DCS) nos permiten realizar un proceso de control unificando la teoría de control y los sistemas de comunicación, siendo capaces de controlar procesos con un elevado número de señales mediante un único bus integrado.

1.2. Antecedentes

El sistema de control distribuido (DCS) surge con el propósito de controlar procesos. Este concepto ha pasado por un largo camino desde los grandes sistemas de control más antiguos hasta los sistemas totalmente escalables de hoy en día que satisfacen una amplia gama de aplicaciones.

Los beneficios que presentan los DCS para el usuario son numerosos, desde una mayor rapidez de la instalación y puesta en marcha, hasta una reducción de los costes de mantenimiento y ciclo de vida.

Este tipo de sistemas de control se han vuelto esenciales en la actualidad en todos los ámbitos que precisen de un control, no solo industrial.

CAN es un protocolo de comunicación basado en la topología buses para la transmisión de mensajes que está regulado por el estándar ISO 11898.

El protocolo CAN trabaja en modo difusión (broadcast) permitiendo la transmisión de paquetes de hasta 8 bits de datos que serán recibidos por todos los dispositivos de la red. Esta característica diferencia a este bus del resto de buses obteniendo numerosas ventajas como una mejor sincronización y un mayor aprovechamiento de los recursos.

Dada sus numerosas ventajas, hemos elegido el CAN en este proyecto para estudiar su uso a nivel de ordenador y a nivel de la BeagleBone Black (BBB), realizando la comunicación entre ambos y llevando a cabo un sistema de control distribuido.

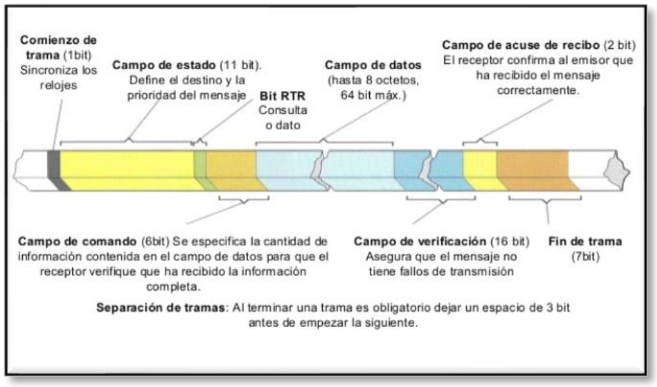

Ilustración 1. Trama CAN [15].

1.3. Motivación

Los motivos principales por los cuales se ha decido llevar a cabo este proyecto ha sido el interés por poder ver como es el funcionamiento real del protocolo de comunicación CAN y desarrollar y comprobar de manera práctica todas las ventajas de las que dispone este bus.

Uno de los alicientes en la elección de este trabajo de fin de grado es haber tenido la oportunidad de poder trabajar con este protocolo en la asignatura de prácticas externas, en la empresa IACTec. Esta primera toma de contacto despertó mi interés sobre el bus CAN y me ayudó a decidir seguir trabajando con él y ver su utilidad a nivel más práctico.

Otro de los aspectos que sirvieron de motivación fue la programación. Realmente ha sido uno de los elementos de la carrera que más me ha costado, pero la importancia de su dominio hoy en día y la necesidad de ingenieros con conocimientos en este ámbito han sido la causa de que, a pesar de las dificultades iniciales que supone la primera toma de contacto con la programación, finalmente me haya decidido a elegir este proyecto para poder seguir aprendiendo y mejorando mis conocimientos.

Para finalizar, está el hecho de poder usar la BeagleBone Black, ya que, si bien físicamente se asemeja al Arduino, es una herramienta que no había utilizado con anterioridad y el poder trabajar con ella y llevar a cabo un sistema de control distribuido junto con el bus CAN me llamaba bastante la atención.

1.4. Objetivos

Los objetivos que se buscan con la realización de este proyecto es adquirir unos conocimientos básicos sobre el protocolo CAN y aprender a programar aplicaciones que usen los interfaces CAN en ordenadores y miniordenadores, especialmente bajo el Sistema Operativo (SO) GNU/Linux, debido a que hoy en día es el más utilizado en sistemas empotrados y de tiempo real.

1.5. Metodología

Para llevar a cabo los objetivos, se deben desarrollar una serie de clases y programas de prueba en C++ que nos permitan trabajar de manera separada por un lado con el protocolo CAN en el PC, y por otro lado con la BeagleBone.

Una vez hayamos realizado las clases y pruebas correspondientes, el siguiente paso será crear una serie de programas que sean capaces de trabajar a la vez con el PC y la BBB transmitiendo y recibiendo mensajes por medio del bus CAN.

Por último, se ha de desarrollar un sistema de control de temperatura mediante un control PID haciendo uso de los módulos PWM y conversor Analógico-Digital de la BBB para poder así implementar el sistema de control distribuido utilizando el protocolo CAN bus.

1.6. Estructura de la memoria

El presente proyecto vendrá estructurado de la siguiente manera:

- En primer lugar, se dispondrá de un capítulo de introducción que permita al lector conocer aquellos aspectos previos y necesarios para poder comprender la elaboración y el funcionamiento de este proyecto.
- En segundo lugar, nos encontraremos con un capítulo de conocimientos previos, que tiene como finalidad explicar una serie de conocimientos básicos sobre los elementos empleados en el proyecto con el fin de facilitar la comprensión y el entendimiento del contenido de este trabajo de fin de grado.
- El tercer capítulo contendrá el desarrollo del proyecto en sí, explicando todos los pasos y tareas llevadas a cabo para realizar los objetivos previstos.
- En cuarto lugar, podemos encontrar las conclusiones obtenidas y posibles líneas futuras que se pueden elaborar.
- Para terminar, el último capítulo, incluirá el presupuesto del proyecto.

Capítulo 2. Conocimientos previos.

2.1. Sistemas de control distribuidos

2.1.1. Los sistemas de control

Un sistema de control se puede definir como el conjunto de dispositivos cuya principal tarea es la administración, dirección y regulación del comportamiento de otro sistema, teniendo como objetivo la obtención de los resultados deseados con la mínima cantidad de fallos posibles. Su uso más habitual se encuentra en los procesos industriales para controlar equipos o máquinas.

Los sistemas de control han ido evolucionando a lo largo de los años, así, hasta los años 60 el más utilizado en la industria era el Sistema de Control Centralizado. Este tipo de sistema se caracteriza por su lógica cableada basada en relés, condensadores y resistencias, transmitiendo señales analógicas llevadas a armario. Dado el elevado uso de cables utilizados para alimentación y señales se daban ciertos problemas como las caídas de tensión, interferencias y complicado mantenimiento.

Para solucionar los problemas mencionados empiezan a surgir dispositivos con microprocesadores, más potentes y con mejores servidores, denominados Sistema de Control Distribuido (DCS).

Ilustración 2. Sistema de Control Centralizado [16].

2.1.2. Los Sistemas de Control Distribuido (DCS).

Un sistema de control distribuido [1] se trata de un sistema de control aplicado a procesos industriales complejos capaz de controlar procesos de hasta 250.000 señales.

Tiene su origen con la aparición de los Controladores Lógicos Programables (PLC), los cuales permiten realizar cambios programables sin producir cambios en el hardware.

La característica principal de un DCS es que posee una única base de datos integrada para todas las señales y variables del sistema, permitiendo que desde un único puesto de control se pueda cargar todos los programas a sus respectivos equipos del sistema. Para ello, todos los equipos deben estar sincronizados bajo un mismo reloj.

De esta manera, la comunicación que se lleva a cabo entre subsistemas es digital y se desarrolla mediante buses de campo (redes digitales, bidireccionales, multipunto montadas sobre un bus serie), lo que nos permite simplificar enormemente la instalación y operación de máquinas utilizadas en la producción ya que realiza la conexión de dispositivos de campo PLCs/PACs, transductores y sensores.

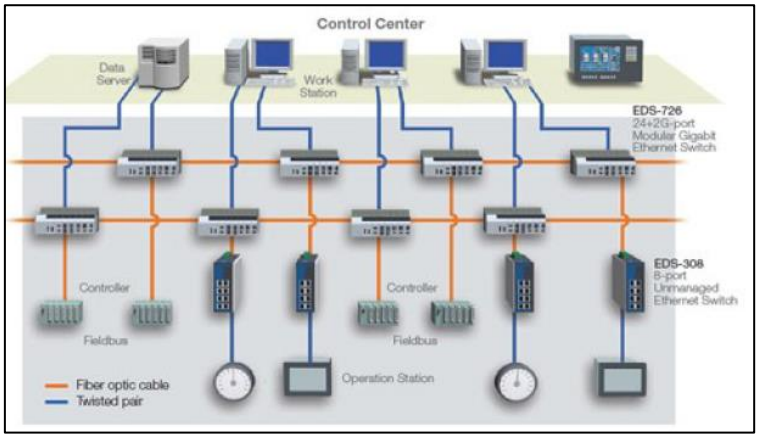

Ilustración 3. Sistema Distribuido por Ordenador [17].

2.2. Bus CAN

El bus CAN [2] (Controller Area Network) es un protocolo de comunicación serie asíncrono del tipo CSMA/CD + AMP (“Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority”).

CSMA significa que cada uno de los nodos de la red debe encargarse de monitorizar el bus y si percibe que no hay ningún tipo de actividad, puede enviar un mensaje.

Mientras que, CD + AMP quiere decir que las colisiones que se den en el bus se resuelven mediante un arbitraje sabio de bits, basado en una preprogramación prioritaria de cada mensaje en el campo identificador del mensaje.

Ambos términos serán explicados en mayor profundidad en el apartado 2.2.4.

La especificación del bus CAN contiene una serie de características interesantes que incluyen:

- La asignación de ancho de banda basado en prioridad. Nos asegura que los mensajes de alta prioridad sean tratados con latencia mínima (retardos temporales).
- Capacidad multimaestro. Se caracteriza porque permite que cualquier nodo que se encuentre en el bus pueda transmitir en cualquier momento que el bus esté libre. Así, cuando dos nodos quieran transmitir al

mismo tiempo el conflicto se resuelve por el arbitraje utilizando el identificador, este arbitraje nos garantiza que no se pierda ni la información ni el tiempo.
Durante el arbitraje los transmisores deben ir comparado el nivel del bit transmitido con el nivel del bus, de manera que, si los niveles son iguales, se pueden realizar envíos, mientras que, si son distintos se pierde el arbitraje y hay que retirarse sin enviar otro bit.

- Retransmisión automática. Las tramas CAN que están dañadas en la transmisión se retransmiten automáticamente. Esto conduce a una capacidad de transferencia muy confiable.
- Seguridad. Para la detección de errores, señalización y auto-chequeo se llevan a cabo ciertas medidas como la señalización del error y del tiempo de restablecimiento que permite que todos los nodos puedan detectar un mensaje corrupto para abortarlo y transferirlo automáticamente con un tiempo de restablecimiento desde la detección del error de 31 bits.

La Ilustración 3 muestra un diagrama de bloques de una red de bus CAN típica. Consiste en dos líneas de señal que reciben el nombre de CANH y CANL. Todos los dispositivos o unidades de control (ECUs) se conectan a las líneas diferenciales y se comunican entre sí a través del bus diferencial. El par de líneas diferenciales que componen el bus lleva en sus extremos una impedancia de 120 Ω para evitar interferencias y ruidos que pueda producir la señal que circula a través de él.

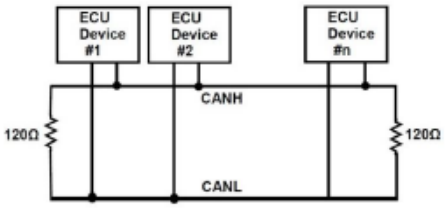

Ilustración 4. Diagrama de bloques de una red CAN típica [18].

Una ECU típica consiste en un transceiver CAN, un controlador CAN y una fuente de reloj.

Las ECU son dispositivos inteligentes y cada ECU tiene la capacidad de transmitir un paquete de datos CAN al bus y recibir paquetes de datos emitidos.

2.2.1. Componentes

- Controlador. Es el encargado de gestionar todas las comunicaciones que se realizan a través del bus, y de acondicionar la información que entra y sale entre el microprocesador y el transmisor-receptor.
- Transceiver. Es transmisor y receptor. Se encarga de transformar los datos del controlador en señales eléctricas y transmitirlas sobre los cables del bus, del mismo modo recibe la información de los cables y la transforma para enviarla al controlador.
- Cables. Proporcionan el medio físico para llevar a cabo una comunicación bidireccional de datos en formato diferencial. Los cables se denominan CANH y CANL y establecen en los nodos los dos niveles lógicos comentados en el apartado 2.2.4.
- Resistencia final de bus. Suele ser de 120 Ω. Proporciona una impedancia controlada que protege de las interferencias por reflexiones de señal.

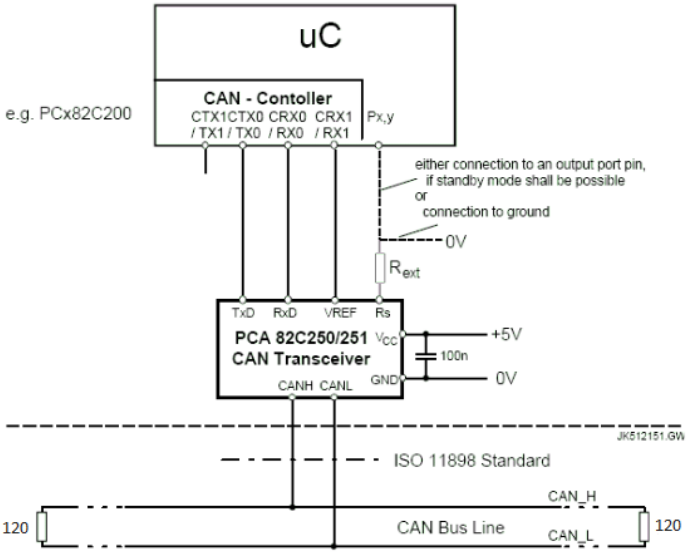

Ilustración 5. Componentes que forman el bus CAN [19].

2.2.2. Norma reguladora

El protocolo de comunicaciones CAN es un protocolo regulado por la norma ISO 11898. Esta norma se publicó como documento en 1993.

Actualmente la norma ISO 11898 se divide en las siguientes partes:

- La parte 1 define la capa de enlace de datos que incluye la subcapa de control de enlace lógico (LLC) y la subcapa de control de acceso al medio (MAC), así como la subcapa de señalización física (PHS).
- La parte 2 define la conexión al medio físico de alta velocidad.
- La parte 3 define la conexión al medio físico tolerante a fallos a baja velocidad (PMA).

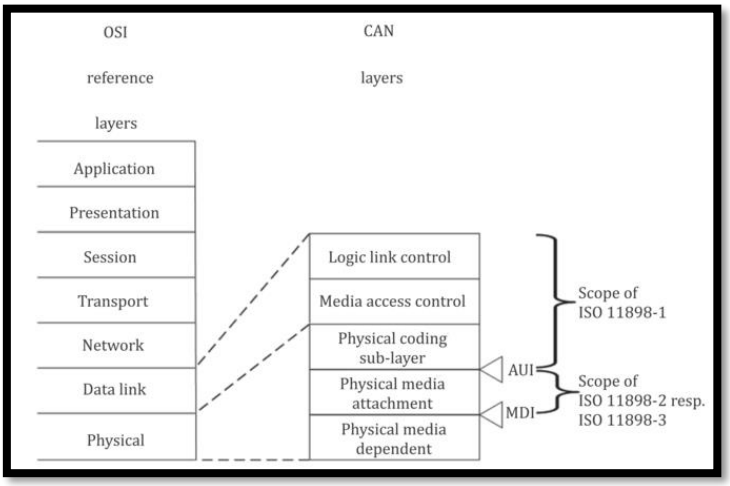

Ilustración 6. Relación capas CAN bus según norma ISO 11898 [20].

2.2.3. Capa física

La capa física es la causante de realizar la transferencia de bits entre los diferentes nodos que componen el sistema, teniendo que definir aspectos como la codificación, los niveles de señal, la sincronización.

Está regulada por el estándar ISO 11898-2. Y está dividida en las siguientes subcapas:

- Señalización física: codificación de bit, temporización y sincronización.
- Unión de medio físico: características de drenaje y recepción.
- Interfaz independiente del medio: conector de bus.

El medio físico empleado se trata de un cable de par trenzado y adaptado en los extremos, permitiendo una velocidad máxima de transmisión de 250 Kbps para la especificación básica y de 1 Mbps para la ampliada.

Este bus permite un número de nodos máximo de 32, transmitiendo entre 2000-5000 mensajes por segundo en un bus de 250 kbps. Por otro lado, la longitud máxima que se permite es de 1 kilómetro, siendo posible añadir dispositivos como repetidores para aumentar el número de nodos que puedan ser conectados.

En cuanto a la topología del bus, los cables, como ya hemos visto, deben ser de par trenzados y su tipología debe estar tan cerca como sea posible a una sola estructura de línea con el objetivo de poder reducir al mínimo las reflexiones que se puedan producir.

La capa física también se encarga de la codificación de bits para la cual se emplea el Método NRZ (Non Return to Zero) que posee las siguientes características:

- No hay flanco de subida o bajada para cada bit.
- Hay estados de bit dominante (Low) y recesivo (High), explicados en el apartado siguiente.
- Permite una disminución de la frecuencia respecto a codificación Manchester.

2.2.4. Capa de enlace

La capa de enlace es la responsable del control lógico y del acceso al medio, está estandarizada por medio de la norma ISO 11898-1, y se divide en las siguientes dos subcapas:

- Control de Enlace Lógico (LLC)
  Se encarga de los filtros de los mensajes, las notificaciones de sobrecarga y la administración de la recuperación. Además, proporciona servicios durante la transferencia de datos y decide que mensajes recibidos del Control de Acceso al Medio (MAC) son aceptados.
- Control de Acceso al Medio(MAC)
  Se encarga de realizar el tramado y destramado de mensajes, de la arbitrariedad a la hora de acceder al bus, del reconocimiento de mensajes y del chequeo de posibles errores. Además, presenta los mensajes recibidos a la subcapa LLC y decide si el bus está libre para comenzar una nueva transmisión.

Como hemos visto, CAN utiliza el control acceso al medio tipo “CSMA/CD+AMP”, dónde las colisiones que puedan surgir (CD) se resuelven mediante la supervivencia de las tramas que chocan en el bus (AMP).

La trama superviviente es aquella que se ha identificado como de mayor prioridad. Para ello, se definen dos niveles, nivel dominante (0) con diferencia de 2 V, y nivel recesivo (1) sin diferencia. Mediante el arbitraje con prioridad de bit, se hace la y-lógica dónde el 0 tiene más prioridad que el 1, y dónde cada nodo escucha a la vez que envía.

Todos los transmisores se quedan en escucha del bus para comparar los valores que circulan con los que cada uno ha enviado, retirándose en caso de que no coincida. Para poder realizar los pasos anteriores de manera correcta hay que aplicar los dos siguientes factores:

- Fijar con claridad la latencia en la transmisión de mensajes.
- Trabajar en modo multimaestro sin necesidad de control de acceso al medio.

El nivel prioritario más alto siempre gana en el acceso al bus, permitiendo que alguno de los mensajes en conflicto termine enviándose, por lo que no hay pérdida de aprovechamiento por colisiones (a diferencia, por ejemplo, del ethernet).

	Lógica	CANH	CANL	Voltaje diferencial
Dominante	0	2.85 V	0.65 V	2.2 V
Recesivo	1	2.3 V	2.3 V	0.2 mV

Tabla 1. Niveles lógicos de los nodos.

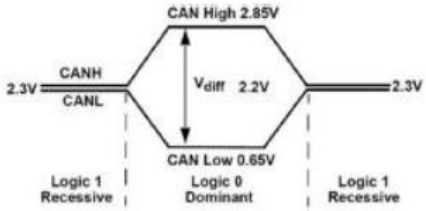

Ilustración 7. Niveles lógicos de los nodos [18].

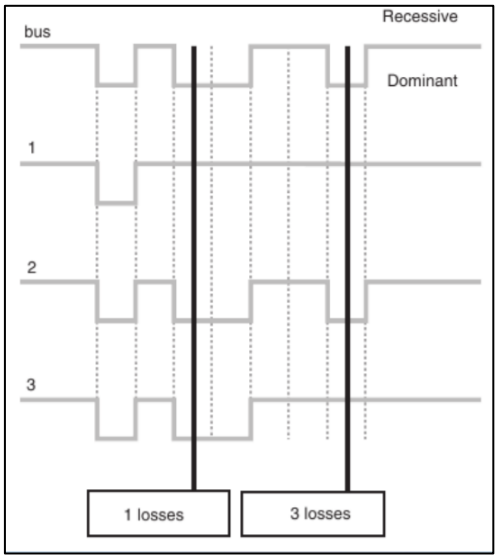

Ilustración 8. Ejemplo arbitrariedad de bits [21].

La unidad básica de transferencia de datos en el bus es la trama CAN. La especificación CAN 2.0 define dos formatos de trama diferentes, uno que utiliza un identificador de trama de 11 bits CAN 2.0A y otro que utiliza un identificador de trama de 29 bits CAN 2.0B. En ambos casos, la cantidad máxima de datos que se pueden transferir por trama es de 8 octetos. Los dos formatos son compatibles en sistemas que cumplan con la especificación CAN 2.0.

2.2.5. Tipo de tramas

En el protocolo CAN existen distintos tipos de tramas predefinidas para la gestión de transferencia de mensaje entre las que se encuentran:

- Trama de datos
  Es la encargada del envío de datos a través de la red, transmitiendo la información útil en modo *broadcast* a todos los nodos de la red. Está formada por:

- Inicio de trama (SOF): está formado por un solo bit en estado siempre dominante que nos indica que se ha iniciado la transmisión.
- Arbitraje (Identifier): lo forman el identificador de mensaje (11 bits (CAN estándar) o 29 bits (CAN extendido), más el bit RTR, que es el encargo de establecer la prioridad de mensaje.
- Control: nos informa de la cantidad de información que está conectada al bus. Está formado por el bit IDE que nos dice si la trama es estándar (IDE dominante) o extendida (IDE recesivo). El segundo bit es el de reserva, r0, que es siempre recesivo. Y para terminar los 4 bits de código de longitud (DLC) que nos muestran el número de bytes de datos del mensaje, de más a menos significativos.
- Datos: lo compone un conjunto de 0 a 8 bytes de datos. Transmite la información y el contenido del mensaje.
- CRC: Código de Redundancia Cíclica formado por 16 bits de verificación. Se encarga de detectar errores en la transmisión de mensajes.
- Campo de reconocimiento (ACK): está formado por 2 bits, el primero de reconocimiento y el segundo para delimitar. El nodo que esté transmitiendo manda una trama con el bit ACK en estado recesivo, mientras que los receptores, si han recibido bien el mensaje, mandan un mensaje en estado dominante.
- Fin de trama (EOF): se encarga de cerrar la trama. Está formado por un total de 10 bit recesivos, 7 bits sucesivos más el espacio entre tramas (IFS) que consta de 3 bits.

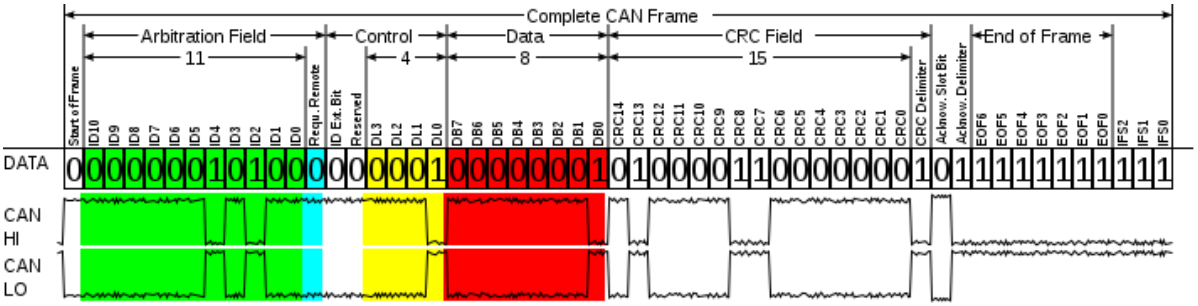

Ilustración 9. Trama de datos para CAN estándar [22].

- Trama remota
  Esta trama es utilizada por un nodo para realizar la solicitud de una trama con un identificador dado para poder obtener los datos que contienen la información requerida.
- Trama error
  Es la encargada de notificar al resto de nodos que se ha detectado error, invalidando el mensaje erróneo.
- Espacio entre tramas
  Es el espacio que separa las tramas entre sí por una secuencia ya establecida.
- Trama de sobrecarga
  Fuerza al resto de nodos a alargar el tiempo de transmisión entre tramas debido a que un nodo se encuentra en sobrecarga.

2.3. BeagleBone Black

BeagleBone [3][4] es una plataforma compacta y económica elaborada por la organización Beagleboad.org que tiene como objetivo principal promover el uso de software y hardware *OpenSource*. Es una tarjeta que trabaja con el sistema operativo Linux, pudiéndose utilizar para desarrollar aplicaciones complejas con interfaces software de alto nivel y circuitos electrónicos de bajo nivel, siendo ideal para la creación de prototipos de proyectos y diseños de productos que permitan sacar ventaja del poder y la libertad de Linux, todo esto combinado con el acceso directo a los pines de entrada/salida y buses, hacen posible poder interactuar con componentes electrónicos, módulos y dispositivos USB.

Como características principales de la BeagleBone podemos destacar:

- Potencia. Contiene un procesador que puede realizar hasta 2 mil millones de instrucciones por segundo.
- Bajo coste. Su precio oscila entre los 50-70 euros.
- Robustez. Soporta muchas interfaces estándar para dispositivos electrónicos.
- Eficiencia. Utiliza poca potencia, trabajando entre 1W y 2’3W.
- Integración. Permite que en una única placa tengamos todo lo necesario, haciendo solo falta alimentarla.
- Libertad. Tiene hardware libre y soporta herramientas y aplicaciones de software libre, y consta de una enorme comunidad de personas innovadoras y entusiastas.

La BeagleBone ejecuta el sistema operativo Linux, lo que significa que se pueden utilizar muchas librerías y aplicaciones de software libre. La disponibilidad del controlador de software *OpenSource* también permite conectar dispositivos tales como cámaras USB, teclados, adaptadores Wi - Fi, etc.

Esta plataforma está formada por la integración de un microprocesador de alto rendimiento en una tarjeta de circuitos impresos (PCB) y un extenso sistema de software. La PCB física no es un circuito completo, más bien es un prototipo de diseño de referencia, que se puede utilizar para construir un circuito completo.

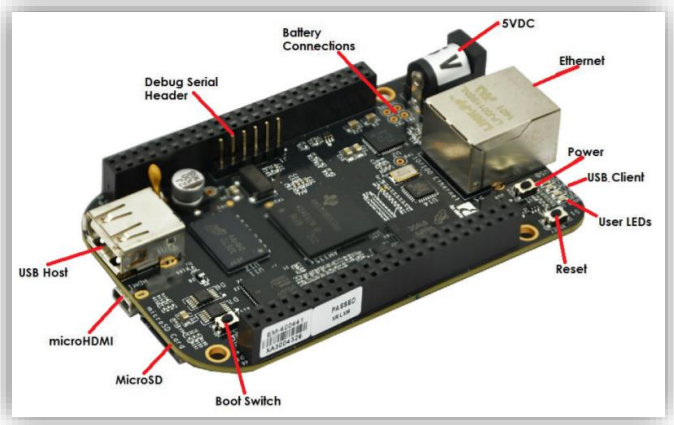

Ilustración 10. Partes de la BeagleBone Black [23].

La BeagleBone Black se caracteriza por el gran número de pines de entrada y salida que dispone (ver ilustración 13). Estos pines presentan numerosas funcionalidades que nos permiten realizar diversas tareas como I/O digitales, salidas con PWM, entradas analógicas. Por otro lado, posee un puerto USB 2.0 para comunicarse con otros dispositivos y un puerto ethernet que permite la comunicación en red con otros dispositivos.. En nuestro caso, utilizaremos aquellos pines que permiten el uso del módulo ADC y del módulo PWM. Además, también

utilizaremos, siendo el núcleo central de este proyecto, uno de los periféricos integrados, que precisamente, es un controlador de bus CAN. Así, en la BBB está el controlador del bus CAN que necesita un transceiver (para generar los voltajes), este transceiver será el adaptador SN65HVD230 explicado en el apartado 2.4.

Una característica impresionante de la BeagleBone es que su funcionalidad puede extenderse a placas secundarias, llamados *capes*, que se conectan a los conectores P8 y P9. De esta manera, podemos diseñar nuestras propias *capes* y añadirlos de forma segura a la BeagleBone utilizando estos conectores. Además, muchos *capes* se pueden comprar para ampliar la funcionalidad de la tarjeta.

Actualmente existen dos versiones de dicha plataforma, la BeagleBone y la Beaglebone Black. Las diferencias entre ambas [5] corresponden sobre todo al hardware. La Black tiene más memoria RAM que la normal, además, esta última no posee memoria incorporada por lo que debe trabajar siempre con una microSD. Por otro lado, la BeagleBone Black posee lo elementos siguientes que no tiene la BeagleBone: eMMC, micro HDMI, encabezado de serie, botón de encendido y botón de inicio de usuario.

Visto que la BeagleBone Black es mucho más completa y está más actualizada, será la que utilicemos. Sus principales características son las siguientes:

	Función	Medio Físico	Detalles
1	**Procesador**	AM335x	Un potente procesador Sitara 1GHz ARM-A8, capaz de procesar 2 billones de instrucciones por segundo
		2 x PRUs	Unidades programables a tiempo real Microcontroladores que permiten una interfaz a tiempo real.
		Tarjeta Gráfica	Tarjeta gráfica 3D (SGX530) capaz de renderizar 20 millones de polígonos por segundo
2	**Gráfica**	HDMI Framer	El framer convierte la interfaz LCD disponible en el procesador AM335x en una señal HDMI (no HDCP)
3	**Memoria SDRAM**	512MB DDR3	La mayoría del sistema de memoria afecta al rendimiento y al tipo de aplicación que queramos ejecutar
4	**Storage**	eMMC (MCC1)	Tarjeta multimedia 2/4 GB integrada incorporada (eMMC) en tarjeta SD o en chip. La BBB puede arrancar sin una tarjeta SD
5	**Fuente de Alimentación**	TPS65217C	Fuente de alimentación IC (PMIC), que tiene 4 reguladores de voltajes LDO. Este IC es controlado por I^2C
6	**Procesador Ethernet**	Ethernet PHY (10/100)	Puede conectarse inmediatamente a una red (aguanta DHCP). La interfaz física LAN8710A conecta el conector físico RJ45 al microprocesador ARM
7	**LEDs**	7 x LEDs	LED de alimentación (azul), 4 LEDs de usuario (azules), y 2 LEDs en el socket RJ45 Ethernet (amarillo = vinculación 100M, verde = tráfico)
8	**Botones**	3 x Botones	Botón de alimentación on/off. Botón de reser para resetear la tarjeta y botón de cambio de arranque para elegir arranque con eMMC o con tarjeta SD

Tabla 2. Características BeagleBone Black.

	Conectores		
9	**Salida de Video**	Micro-HDMI (HDMI-D)	Para conectarse al monitor o televisiones. Soporta resoluciones de 1280x1024 a 60Hz. Puede ir también a 1920x1080 pero solo a 24Hz. Tiene un soporte HDMI CEC
		Audio Out (HDMI-D)	Salida de audio para conectar diversos dispositivos.
10	**Network**	Ethernet (RJ45)	10/100 Ethernet mediante conector RJ45. No tiene Wi-Fi incorporado.
11	**Alimentación DC**	5V DC Supply (5.5 mm)	Alimentación en corriente continua de 5 V
12	**Tarjeta SD**	Card siot (MMC0) (micro-SD)	Ranura de tarjetas micro-SD de 3.3V. Esta ranura se puede inciar, flashear, o usar para otro almacenamiento cuando se arranca con eMMC
13	**Serial Debug**	6 Pin Connector (6 x 0.1”)	(UART0) Se utiliza con un cable serial TTL3V3 para conectar la consola seria de la BBB
14	**USB**	1 x USB 2.0 Client (mini-USB)	(USB0) Conectarse con el escritorio del PC y puede alimentar la BBB directamente y/o comunicarse
15	**USB**	1 x USB 2.0 Host (USB-A)	(USB1) Puedes conectar los periféricos USB con la BBB con este conector USB.
16	**P8 & P9 Cabezales Expansión**	Two 2x23 pin 0.1” female headers	92 pines en dos cabezados que están multiplexados para mejorar el acceso. No todas las utilidades están disponibles al mismo tiempo
17	**Otro Debug**	JTAG	Hay un espacio para un conector JTAG en la capa bottom de la tarjeta. JTAG permite depurar la tarjeta, pero requiere hardware y software adicional
18	**Otra Alimentación**	Battery Conecctors	Es posible soldar pines y utilizarlos para conectar una alimentación adicional

Tabla 3. Conectores BeagleBone Black.

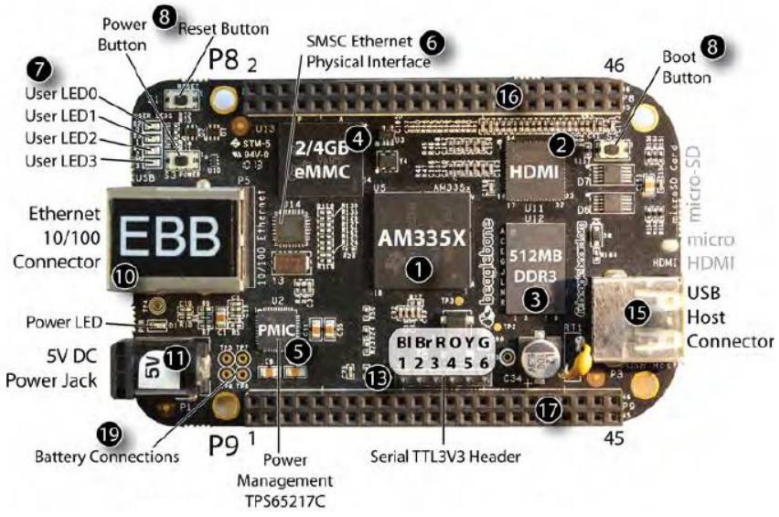

Ilustración 11. Señalización características BeagleBone Black [24].

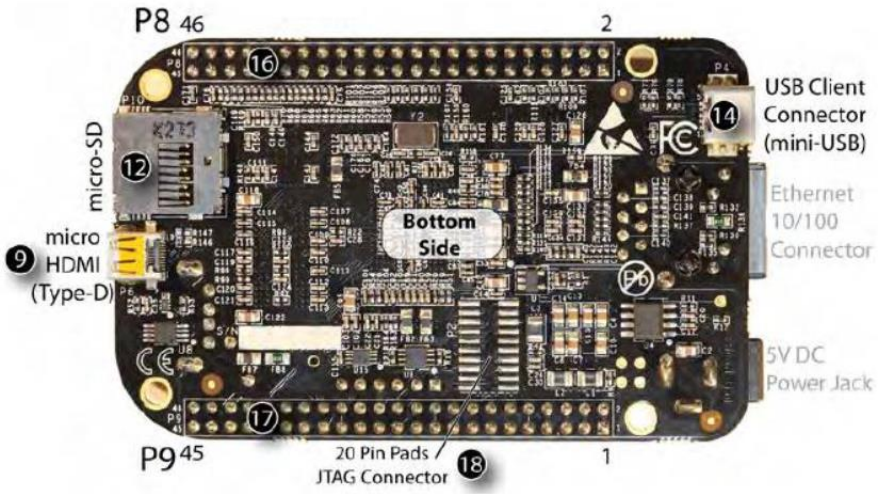

Ilustración 12. Señalización características BeagleBone Black [24].

La forma para acceder a los distintos dispositivos que ofrece la BeagleBone Black ha cambiado fuertemente en los últimos años.

En un primer momento había soluciones para placas concretas (BBB, RaspBerryPi, etc.) Pero en los últimos años se ha hecho un esfuerzo en el Kernel de Linux para homogeneizar el acceso a los dispositivos entre todas las placas, de manera que podamos pasar de una a otra sin esfuerzo.

Por ese motivo las librerías desarrolladas han quedado obsoletas y sólo sirven para unas versiones anteriores del Kernel (3.8, 4.4) pero no para el actual 4.9.

La mejor solución en nuestro caso es hacer nuestro propio código para el acceso a través de *sysfs* (sistema de archivos virtual disponible en */sys*).

Como ya hemos comentado, esta plataforma dispone de varios módulos que nos permiten trabajar diferentes aspectos y aplicaciones. Los módulos de los que haremos uso en el proyecto son:

2.3.1. Módulo ADC

El AM335x tiene un registro de aproximación (SAR) de 12 bits sucesivos ADC que es capaz de obtener 200.000 muestras por segundo. La entrada al SAR se selecciona internamente mediante un conmutador analógico 8:1, y la BBB hace disponibles siete de estas entradas conmutadas en el encabezado P9 como entradas ADC, además de los voltajes de referencia del conversor VDD_ADC y GND_ADC. Recordando que el conversor trabaja en el rango de 0 a 1.8 V.

Las entradas analógicas se pueden configurar para ser utilizadas de varias maneras, en nuestro caso, las entradas ADC se utilizan como entradas simples de un cable ADC. De forma predeterminada, estas entradas no están habilitadas en la BBB, por lo que el primer paso es habilitarlas como se explica en el apartado 3.5.1.

2.3.2. Módulo PWM

El módulo de PWM de la BeagleBone Black cuenta con ocho salidas PWM asociadas a 5 periféricos internos, tres módulos eHRPWM (dos salidas cada uno), y dos módulos eCAP.

Las ocho salidas eHRPWM están disponibles en dos clavijas separadas (por ejemplo, la salida 2B está disponible en P8_13 y P8_46). Al igual que la funcionalidad ADC, los pines PWM están habilitados mediante superposiciones, por lo que antes de utilizaros debemos habilitarlos como se explica en el apartado 3.6.2.

La configuración de la PWM la haremos a través del *sysfs* en el */sys/class/pwm/pwmchipN* siendo la relación de los pines la siguiente:

Chip	Pines	Canal
pwmchip0	P9_42	canal 0
pwmchip1	P9_22	canal 0
	P9_21	canal 1
pwmchip3	P9_14	canal 0
	P9_16	canal 1
pwmchip5	P9_28	canal 0
pwmchip6	P8_19	canal 0
	P8_13	canal 1

Tabla 4. Configuración pines BBB.

En la siguiente ilustración, se muestran todos los pines que posee la BeagleBone Black resaltando aquellos que vamos a usar:

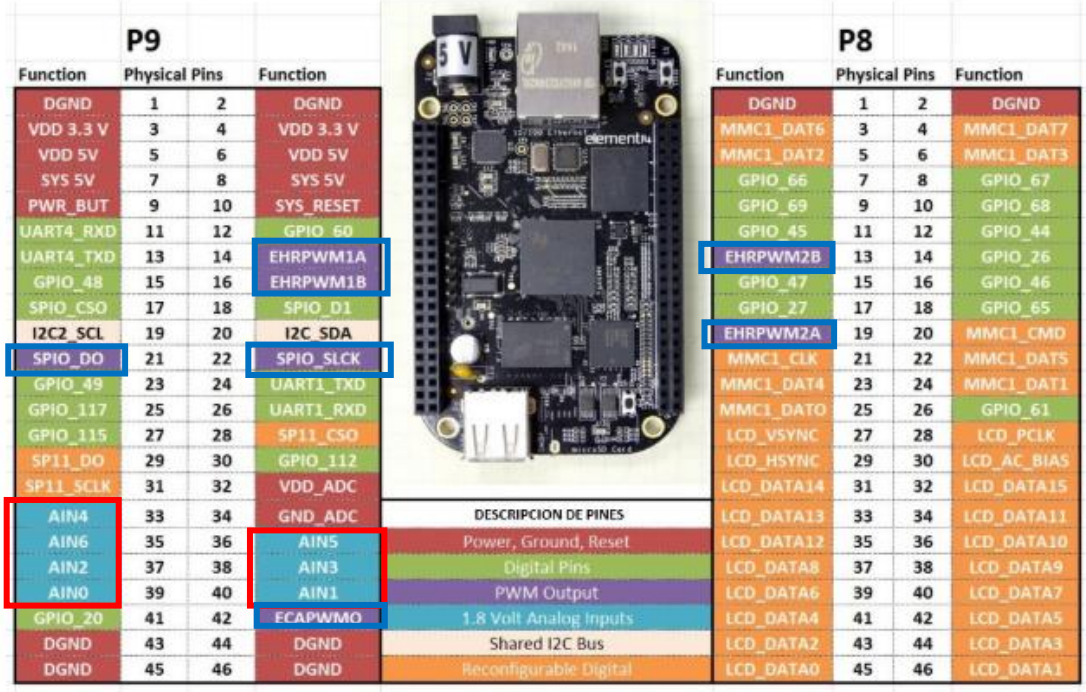

Ilustración 13. Número de path de la BeagleBone Black [25].

2.4. Adaptador basado en SN65HVD230

Se utiliza para poder establecer la comunicación CAN entre la BeagleBone Black y el bus CAN con todo lo que hay conectado a él.

Las características principales del adaptador son:

- Capacidad para ser utilizado en alta interferencia.

- Totalmente compatible con las normas ISO 11898.
- Alta impedancia de entrada.
- Modo de espera a baja corriente.
- Velocidad de referencia de hasta 1 MB/s.
- Protección térmica.

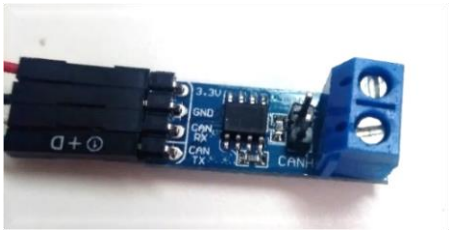

Ilustración 14. Adaptador basado en SN65HVD230.

2.5. Tarjeta CPC_PCI

CPC_PCI es una placa de plug-in CAN pasivo para ranuras PCI. CPC_PCI fue diseñado para la industria en serie y tiene una construcción robusta y rentable. CPC_PCI soporta uno o dos canales CAN que pueden ser operados independiente con diferentes velocidades de datos. El interfaz viene con el chip SJA1000 del controlador Philips CAN, que ofrece buenas cualidades de diagnóstico.

La CPC_PCI permite también dotar a un PC convencional con bus PCI de uno (o dos) periféricos de conexión al bus CAN.

CPC_PCI mapea los controladores CAN directamente en el espacio de direcciones del PC y permite el acceso a los mensajes CAN con bajas latencias. El software existente que posee el controlador CAN puede ser fácilmente adaptado. La comunicación CAN con CPC_PCI puede ser manejada ya sea en modo controlado por interrupción o en modo sondeado, los canales de interrupción se asignan automáticamente (Plug & Play).

Si se quiere realizar la instalación de los drivers adecuados podemos acceder a la página oficial de Can4Linux [8] dónde podemos encontrar toda la información acerca del driver para el sistema GNU/Linux. Si se quiere descargar el paquete acceda al enlace de la biografía [8] y a su página web oficial [9]. Pero actualmente con la aparición del socketCan esto no es necesario ya que en los kernels nuevos viene por defecto y las tarjetas lo detectan directamente.

Ilustración 15. Tarjeta CPC_PCI.

2.6. Controlador PID

2.6.1. Introducción

Un controlador PID [13] [14] (Proporcional-Integral-Derivativo) se trata de un mecanismo de control retroalimentado en un bucle cerrado que se usa en sistemas de control industrial.

Su pincipal aportación es que nos permite calcular un error como la diferencia entre el valor actual y el valor al que queremos llegar intentando minimizar el error ajustando las entradas del proceso.

Para el calculo de un controlador PID [13] intervienen tres acciones diferentes. En primer lugar, la proporcional P que depende del error actual, la integral I que es la suma de todos los errores pasados y la derivativa D que predice los errores futuros según la tasa de cambio actual. Para ajustar un proceso mediante un elemento de control se realiza la suma de estos tres términos.

$$U = P + I + D \tag{2.1}$$

Un aspecto importante que debemos tener en cuenta es que el uso de un controlador PID no nos garantiza un control impecable ni una estabilidad en el sistema.

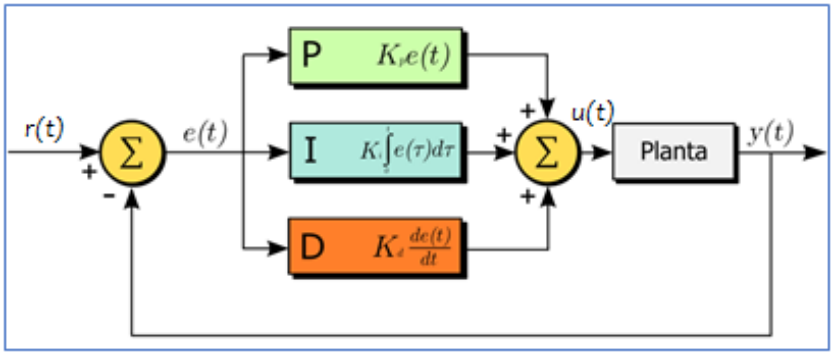

Ilustración 16. Diagrama de bloques de un controlador PID [26].

2.6.2. Acciones de un controlador PID

2.6.2.1 Acción Proporcional

La parte proporcional se encarga de modificar la salida proporcionalmente con el error actual. La respuesta del sistema obtenida se ajusta multiplicando el error por la ganancia proporcional (Kp). Lo que se busca es que el error en estado estacionario se aproxime a cero.

$$P = K_p \cdot e(t) \tag{2.2}$$

Este tipo de controlador para la mayoría de sistemas no nos garantiza que el objetivo se cumpla 100% ya que puede conservar un error en el estado estacionario.

2.6.2.2 Acción Integral

El objetivo de la acción integral es disminuir y eliminar el error en estado estacionario llegando antes al valor deseado.

$$I = K_i \cdot \int_0^t e(t)dt \tag{2.3}$$

La acción integral actúa cuando se produce una desviación entre la variable y el punto de consigna. Tiene en cuenta los errores acumulados en el pasado lo cual puede provocar que el valor actual sobrepase el valor deseado.

2.6.2.3 Acción Derivativa

La acción derivativa se da cuando hay un cambio en el error, se encarga de calcular la variación del error mediante la pendiente del error en cada instante de tiempo, es decir, se deriva con respecto al tiempo y se multiplica por una constante Kd, denominada ganancia derivativa.

$$D = K_d \cdot \frac{d}{dt} e(t) \tag{2.4}$$

La acción derivativa es utilizada para anticiparse en el tiempo con el objetivo de evitar los sobrepasamientos.

2.6.3. Control PI

El control PI es un control que utiliza únicamente la acción proporcional y la acción integral, con el propósito de obtener una respuesta estable y sin error en régimen permanente.

$$u(t) = K_p \cdot \left(e(t) + \frac{1}{T_i} \int_0^t e(t)\, dt \right) \tag{2.5}$$

2.6.4. Control PD

El control PD es un control que trabaja con la acción proporcional y la acción derivativa prescindiendo de la acción integral. Se utiliza cuando no es necesario un error nulo en régimen permanente.

$$u(t) = K_p \cdot \left(e(t) + T_d \frac{de(t)}{dt} \right) \tag{2.6}$$

2.6.5. Control PID

En el control PID intervienen las tres acciones, como hemos comentado anteriormente, las tres acciones se suman para obtener la salida del controlador PID, de esta manera, si denominamos a la salida como u(t), la ecuación que la define es:

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(t)\, dt + K_d \cdot \frac{d}{dt} e(t) \tag{2.7}$$

Dado que vamos a llevar a cabo el control PID por medio de un ordenador la ecuación anterior no es la más idónea, esto se debe a que en un sistema digital [14] los valores se evaluan o cambian cada cierto periodo de tiempo, este periodo se conoce como tiempo de muestro, que define cuantas veces por segundo se va a realizar las conversiones ADC y se va a obtener la salida del PID, por ello debemos discretizar la ecuación:

$$u(t) = K_p \cdot e_k + K_i \cdot T_s \cdot \sum_{j=0}^{k} e_k + K_d \cdot \frac{e_k - e_{k-1}}{T_s} \tag{2.8}$$

Dónde Ts es el periodo de muestreo.

Una vez se ha discretizado tenemos que calcular el valor de la acción de control para un instante de tiempo. Cuando la hayamos obtenido, volvemos a calcular la acción de control para el siguiente instante de tiempo utilizando la nueva salida del sistema, lo que permite al sistema ir avanzando hacia el estado deseado.

2.7. Sistema Operativo GNU/Linux

El sistema operativo GNU/Linux es un sistema operativo que está dentro de lo que se denomina software libre.

Este tipo de software se caracteriza por cumplir las 4 libertades:

- Libertad para ser usado. Lo puede utilizar cualquier persona, empresa, etc.
- Libertad para copiarlo. Podemos hacerle una copia.
- Libertad para modificarlo. Debe tener disponible el código fuente.
- Libertad para distribuirlo. Tal cual lo hemos recibido o con las modificaciones que hemos hecho.

El software libre se desarrolla a través del proyecto GNU iniciado en 1984 por Richard Stallman quien tiene como objetivo construir un Sistema Operativo Libre, para ello, fundó la Free Software Foundation para apoyar este proyecto.

La combinación del proyecto GNU y el núcleo Linux dan lugar al sistema operativo GNU/Linux, que es el utilizado en este proyecto. Se trata de un sistema operativo multitarea y multiusuario que comprende el uso de las 4 libertades, dónde son los propios usuarios lo que ayudan a su mejora y desarrollo.

2.8. Uso del bus CAN en Linux

Can4linux [8] [9] es un driver de dispositivos *OpenSource* para el kernel de Linux. El inicio de su desarrollo se da a mediados de la década de los 90, siendo en 1995 cuando se creó la primera versión para utilizar el bus CAN con Linux.

El avance de la tecnología ha provocado que Can4Linux se haya quedado obsoleto en el manejo de tarjetas CAN, dando paso al socketCan [6] [7] que se convierte en la nueva forma de manejo de estas tarjetas. En los kernels nuevos ya viene por defecto y las tarjetas lo detectan directamente.

El paquete socketCan es una implementación de protocolos CAN para Linux que utiliza el Berkeley socket API, la pila de red de Linux e implementa los drivers del dispositivo CAN como interfaces de red.

Fue diseñado para superar las limitaciones que nos daban las versiones antiguas, tales como que solo se pueda abrir un único dispositivo al a vez, que el intercambio del controlador CAN necesita de otro dispositivo driver, etc.

Para poder llevar a cabo nuestro proyecto es necesario realizar una serie de configuraciones de la tarjeta CPC-PCI para que esta puede ser utilizada del mismo modo que una tarjeta de red.

Lo primero que debemos hacer es establecer el *bitrate* y levantar la tarjeta. Para ello desde el terminal ejecutamos los siguientes comandos:

- Bitrate → *sudo ip link set can0 type can bitrate 125000*
- Levantamiento → *sudo ifconfig can0 up*
- Comprobar estado → *ip details link show can0*

Este paso también se debe realizar desde la BeagleBone Black, donde debemos configurar los pines a utilizar y luego establecer el *bitrate* y llevar a cabo el levantamiento. Para ello, por ejemplo, en el caso de la tarjeta can0 introducimos los siguientes comandos:

- Configuración de pin → *config-pin P9_19 can*

- Configuración de pin → *config-pin P9_20 can*
- Bitrate → *sudo ip link set can0 type can bitrate 125000*
- Levantamiento → *sudo ifconfig can0 up*

Además, debemos tener en cuenta la siguiente librería que permiten la utilización de CAN desde el PC:

- Librería linux, compuesta por los ficheros can.h , can/raw.h y can/error.h.

2.9. C++

C++ [28] es un lenguaje de programación de alto nivel. El desarrollo de este lenguaje se inicia por Bjarne Stroustrup en 1979 bajo el nombre “C con clases”. Como su propio nombre indica se basa en el lenguaje C tradicional, pero incorpora la programación orientada a objetos.

La idea de Stroustrup era crear un lenguaje que combinara el poder y la eficiencia de C con abstracciones de alto nivel para gestionar mejor los grandes proyectos de desarrollo. Así, en 1983 “C con clases” sufrió un cambio de nombre al ya conocido C++ (C-Plus-Plus), el cual sigue manteniendo compatibilidad con C permitiendo que la mayoría de los códigos programados en C se puedan compilar fácilmente en C++.

La creación e introducción de C++ fue un paso importante en la industria del software debido a que permitía el control del hardware por parte del software tanto en microprocesadores como en sistemas empotrados. Además, C++ está considerado como uno de los leguajes más rápidos y está muy cerca de los lenguajes de bajo nivel, lo que permite un control completo sobre la asignación y administración de la memoria. Esta característica y sus muchas otras capacidades también lo convierten en uno de los lenguajes más difíciles de aprender y manejar a gran escala.

Hay varias razones para la elección de C++. La razón principal fue la rara combinación de abstracciones de alto nivel y bajo nivel del hardware. La eficacia del bajo-nivel fue heredada de C, y las construcciones de alto nivel vinieron en parte de un lenguaje de simulación llamado Simula. Esta combinación permite escribir software C++ con la fuerza de ambos enfoques. Otro punto fuerte del lenguaje es que no impone un paradigma de programación específico a sus usuarios. Está diseñado para dar al programador una gran cantidad de libertad al apoyar muchos estilos de programación diferentes.

C++ es un lenguaje que está en continua evolución y es actualizado por el comité de estándares de C++. Así, en 1998, se publicó el primer estándar internacional, conocido como C++ 98. Desde entonces, este lenguaje ha sufrido tres versiones más con extensiones adicionales, incluyendo C++ 03, C++ 11, y más recientemente, C++ 14, que es el último estándar ISO.

He de añadir que C++ cuenta con una página que nos permite buscar información acerca de alguna librería o algún comando que queramos utilizar, esta página corresponde con el enlace [12] de la bibliografía y se ha utilizado en el desarrollo de este proyecto.

Capítulo 3. Desarrollo del proyecto

El objetivo principal de este capítulo es explicar todo el desarrollo del proyecto paso a paso, comentando de manera detallada cada una de las tareas realizadas para poder cumplir con el objetivo propuesto.

3.1. Introducción al proyecto

Se pretende llevar a cabo una serie de clases y programas en C++ que permitan trabajar con el bus CAN en diversos dispositivos. Todos los programas creados se ciñen al estándar C++ 14 y para la gestión de errores las clases lanzan excepciones.

En primer lugar, se trabajará únicamente con el bus CAN en el ordenador que consta del sistema operativo GNU/Linux. Una vez se haya conseguido controlar el funcionamiento del bus CAN en el PC enviando y recibiendo tramas, se realizarán una serie de clases para poder obtener parámetros en la BeagleBone Black, utilizando su módulo ADC y PWM. De esta manera, cuando ya se haya llevado a cabo el control de ambos dispositivos, el siguiente paso es conseguir que ambos se comuniquen entre sí por medio del bus CAN, para finalmente poder elaborar un control PID de temperatura realizando un control distribuido de todo el sistema.

3.2. Montaje realizado y materiales empleados

3.2.1. Montaje general

Es necesario comentar que todo el material utilizado en la realización de este proyecto procede del laboratorio de la Universidad de La Laguna y proporcionado por el tutor de este Trabajo de Fin de Grado Alberto F. Hamilton Castro.

Los materiales empleados van aumentando o disminuyendo según la parte del proyecto en la que estemos. A continuación, se muestra un montaje general en el que se utilizan todos los materiales necesarios para la realización del proyecto:

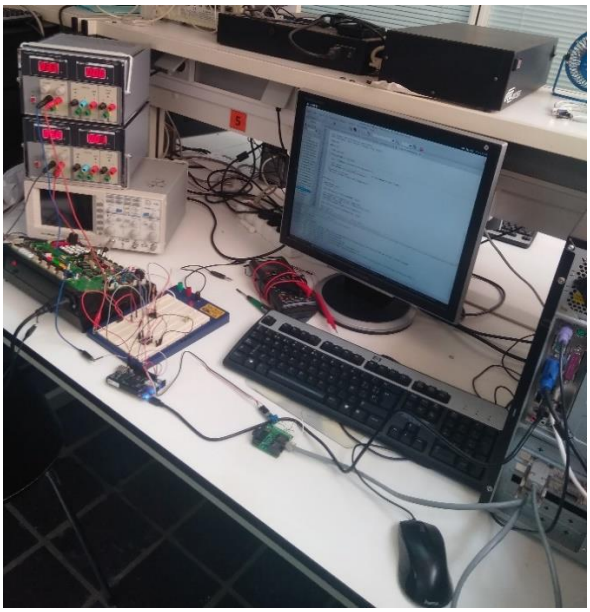

Ilustración 17. Montaje del proyecto.

Se dispone de un ordenador dónde se realizará la programación necesaria y el cual dispone de las tarjetas CAN para poder establecer comunicaciones. Además, nos encontramos también con la BeagleBone Black, la placa de experimentos DISEN-EXP, las fuentes de alimentación, un multímetro, una protoboard para llevar a cabo los circuitos, un osciloscopio que nos permite visualizar distintas señales, el adaptador basado en SN65HVD230 para comunicar la BBB con el ordenador, y los distintos componentes tales como resistencias, un operacional, LED, un potenciómetro y cables necesarios para construir los circuitos.

3.2.2. Ordenador

El ordenador que se ha utilizado dispone de las siguientes características:

Características	
Sistema Operativo	Xubuntu 14.04
CPU	PentiumR Dual-Core CPU E5200 @ 250GHz
Memoria RAM	3943 MiB
Storage	40 GB

Tabla 5. Características del ordenador del laboratorio.

El PC es empleado para llevar a cabo todos los programas necesarios para la realización del proyecto. La programación se hará en el editor de texto Geany y con el lenguaje de programación C++.

Por otro lado, nos permite manejar el terminal de la BBB así como su propio terminal para realizar las comunicaciones.

Tiene instalado dos tarjetas CAN que son dispositivos que implementan el protocolo del bus CAN (transmiten y reciben), dónde el driver del socketCan se carga automáticamente como se muestra en la imagen siguiente:

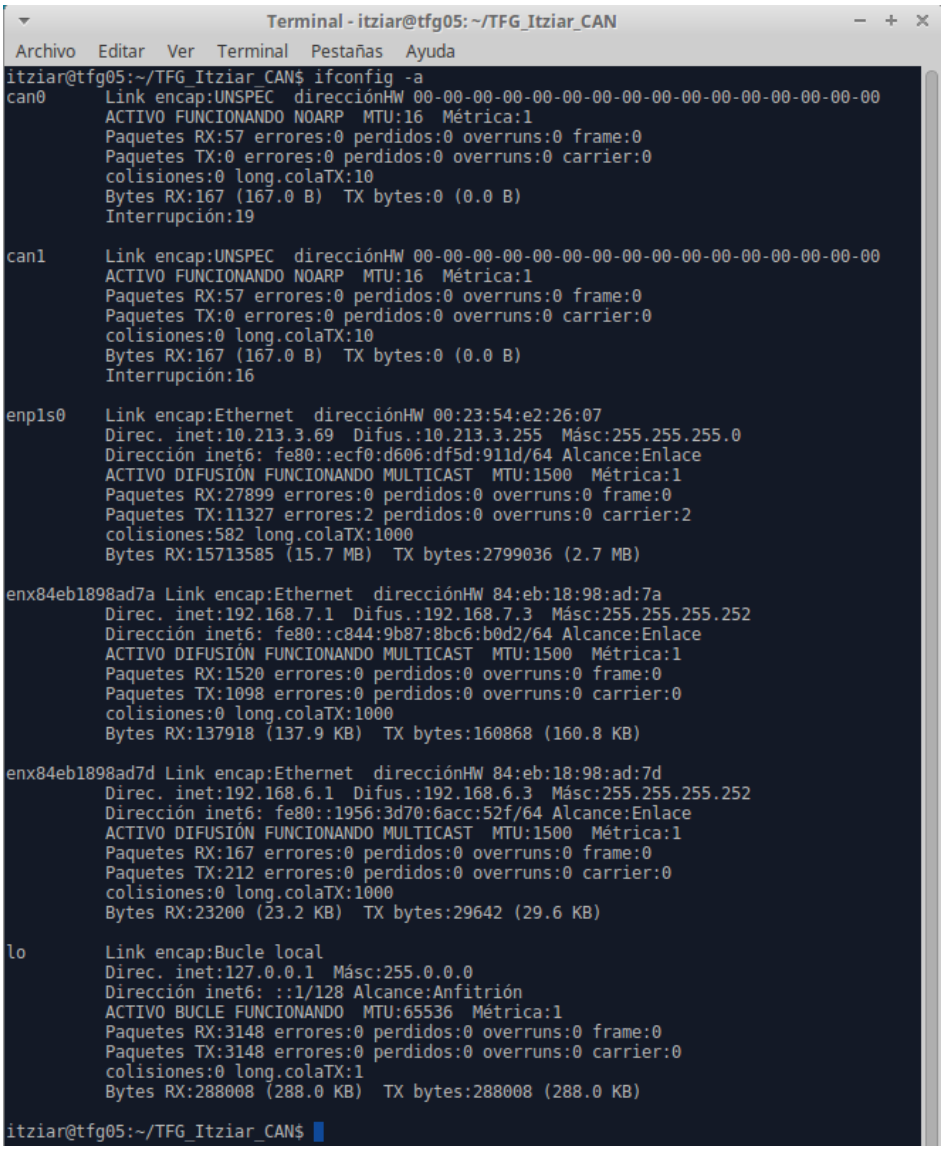

Ilustración 18. Driver socketCan.

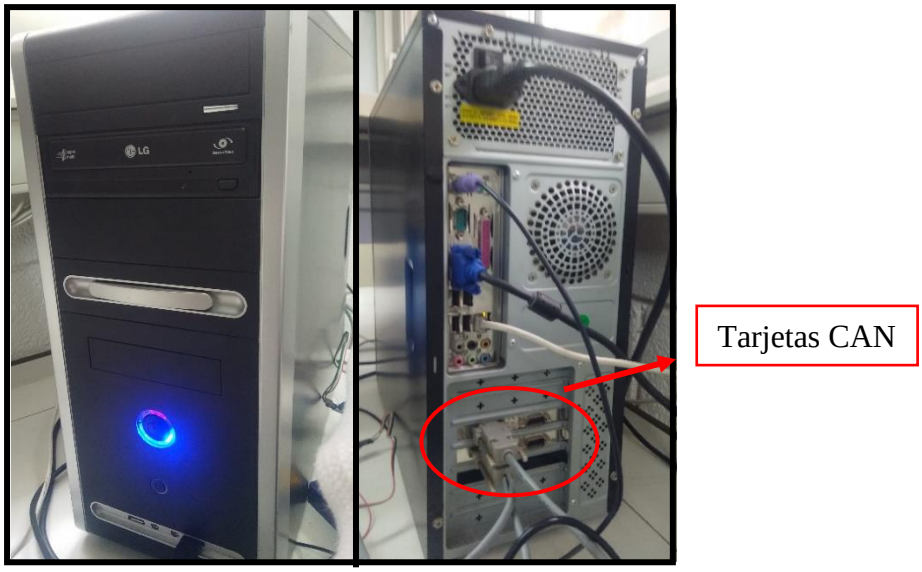

Ilustración 19. Torre del ordenador.

Ilustración 20. Pantalla del ordenador.

3.2.3. Tarjeta CAN

Es el dispositivo encargado de conectarse al bus para recibir y transmitir mensajes. Está incorporada en el ordenador y configurada mediante el driver socketCan para poder funcionar en el sistema operativo GNU.

3.2.4. Convertidor de nivel

Este dispositivo nos permite cambiar de una tensión de 3.3 V a 5 V y viceversa. En nuestro proyecto lo utilizaremos para aumentar el voltaje que nos proporciona la PWM de la BBB y poder trabajar con la placa DISEN-EXP que trabaja con un voltaje de 5Vcc.

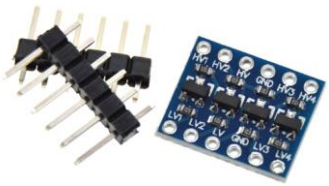

Ilustración 21. Convertidor de nivel [27].

3.2.5. Protoboard

Se trata una placa con agujeros conectados eléctricamente entre sí, en la cual se permite la inserción de componentes electrónicos y cables para la realización de circuitos electrónicos.

Realizaremos tres circuitos sobre ella, uno para medir aspectos analógicos, otro para trabajar con la modulación PWM y el último para el control del PID.

Ilustración 22. Protoboard.

3.2.6. Componentes electrónicos y cables

Los componentes que hemos utilizado son los siguientes:

- Un potenciómetro de 1KΩ para trabajar con las entradas analógicas de la BBB.
- Dos resistencias de 330Ω utilizada en el montaje de circuitos para PWM y PID.
- Un led utilizado en el montaje de circuitos para PWM.
- Un operacional LM358 para el control del PID.
- Cables para realizar las diferentes conexiones.

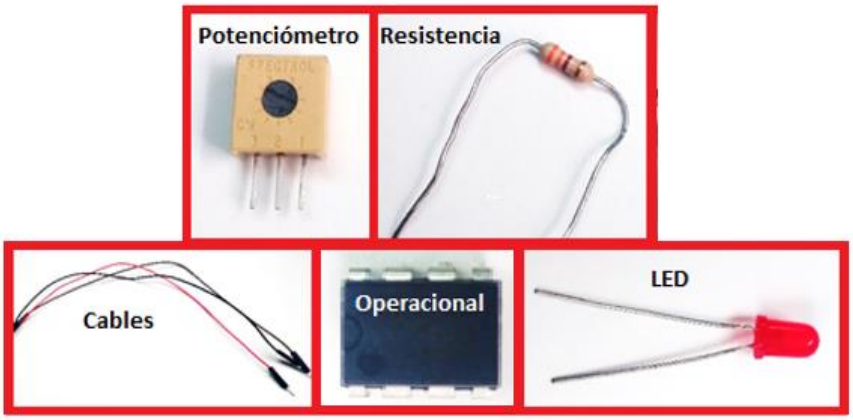

Ilustración 23. Componentes.

3.2.7. Placa DISEN-EXP

Se trata de una placa de experimentos que consta con numerosos circuitos para poder realizar un gran número de pruebas de todo tipo. Desde el control de una cuenta en lenguaje ensamblador, hasta el control PID de temperatura o velocidades.

En nuestro caso utilizaremos el circuito marcado en la imagen siguiente para hacer su control de temperatura por medio de un regulador PID.

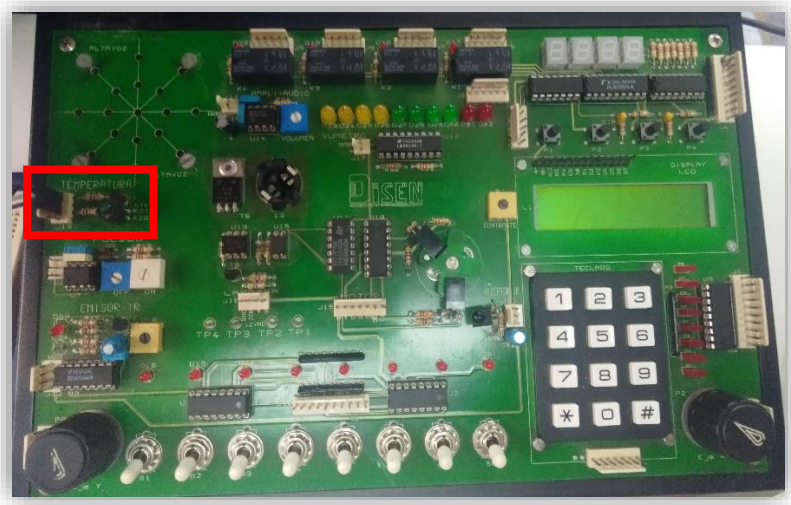

Ilustración 24. Placa DISE-EXP.

3.2.8. Fuente de alimentación

Provee de tensión nuestros circuitos, ya que se encarga de transformar la corriente alterna en corriente continua para que la podamos utilizar.

Ilustración 25. Fuente de alimentación.

3.2.9. Multímetro

Nos permite medir la corriente y tensión entre dos puntos de nuestros circuitos. Además, podemos obtener valores de resistencia y de temperatura.

Ilustración 26. Multímetro.

3.2.10. Osciloscopio y sondas

Un osciloscopio es un instrumento empleado para la visualización gráfica de señales eléctricas que pueden variar en el tiempo. Su uso en este proyecto consiste en ir viendo los diferentes comportamientos de la señal PWM según vamos cambiando los valores del período o del porcentaje de duty cycle.

Además, para poder visualizar la señal de nuestro circuito se hará la comunicación entre ambos por medio de una sonda.

Ilustración 27. Osciloscopio.

3.3. Manejo del bus CAN desde GNU/Linux

Antes de meternos de lleno en la explicación de la programación, se aclara que todos los códigos desarrollados se pueden encontrar en esta página web [11]. En cada una de las explicaciones se comentará a que programa de ese link estamos haciendo referencia.

Se dispone también, en el apartado 3.11 de la tabla 10 que nos indica el identificador de cada una de las tramas que se irán enviando a través del bus a lo largo del proyecto.

Para poder utilizar el bus CAN desde nuestro ordenador lo primero que tenemos que hacer es descargar el paquete que contiene las diferentes utilidades que nos permiten su uso y, con las cuales trabajaremos a lo largo del proyecto. Para descárgalo debemos ejecutar el siguiente comando:

- *apt-get install can-utils*

Dentro del paquete *can-utils* [10] hay diferentes herramientas para estudiar el CAN, las que vamos a usar son las siguientes:

- *cansend*
  Es el encargado de enviar tramas por el bus CAN en un terminal Linux. La sintaxis que utilizamos para ejecutar el comando es la siguiente:
  - Nombre de la tarjeta CAN por dónde se quiere enviar la trama (can0, can1, …).
  - Un identificador, de 11 (can estándar) a 29 (can extendido) bits escrita en hexadecimal.
  - Un símbolo separador entre cabecera y trama (#).
  - Los datos que se desean enviar, escrita en hexadecimal y con un máximo de 8 bytes, es decir, 16 dígitos

  Un ejemplo sería: *cansend can0 1A3#0102030405060708*

- *candump*
  Se utiliza para que el programa se quede en espera de recepción de cualquiera de las tramas que circulan por el bus y las muestra por pantalla, sin importar el tamaño del identificador, los filtros de aceptación, etc.

Estos comandos se pueden utilizar simultáneamente gracias a que el ordenador dispone de dos placas CPC_PCI representando, cada una, un dispositivo diferente permitiéndonos utilizar en una de ellas el comando *cansend* y en la otra el *candump*.

En el inicio del proyecto se comprobó la utilidad de estos dos comandos para verificar que los cables de bus y las tarjetas CPC_PCI funcionaban correctamente. Para ello, se abrieron dos terminales desde el PC dónde en uno, dejábamos uno de los dispositivos en escucha, y en el otro, realizábamos el envío como se muestra en las siguientes imágenes:

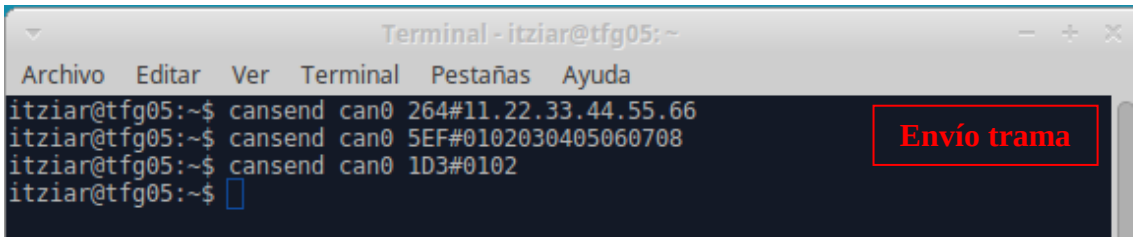

Ilustración 28. Dispositivo can0 enviando trama por el bus.

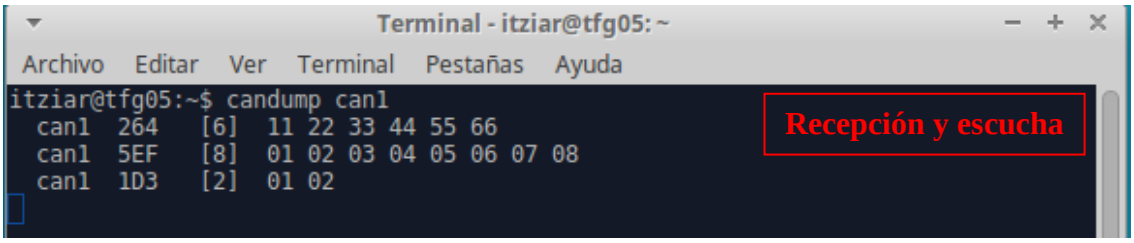

Ilustración 29. Dispositivo can1 recibiendo trama de can0 y en espera de más tramas.

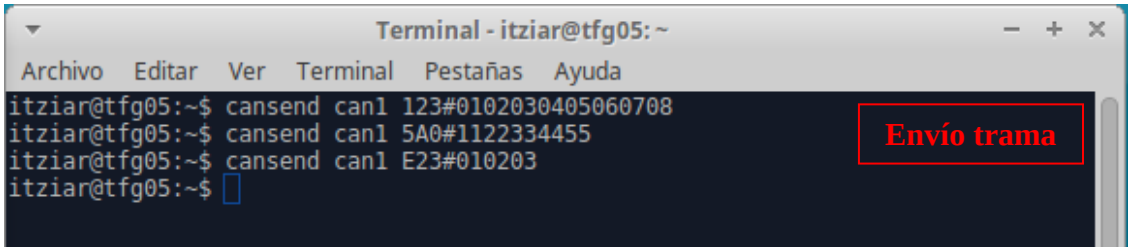

Ilustración 30.Dispositivo can1 enviando trama por el bus.

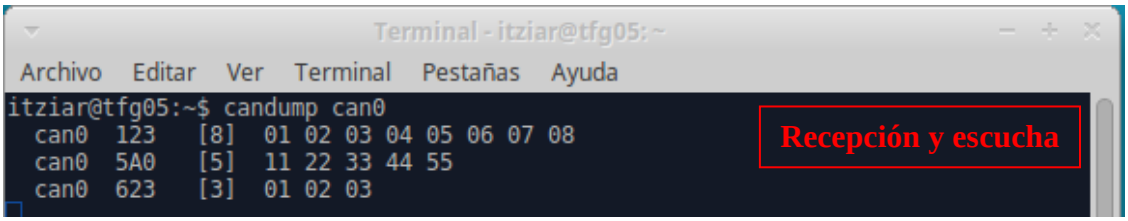

Ilustración 31. Dispositivo can0 recibiendo trama de can1 y en espera de más tramas.

Una vez comprobamos que el envío y la recepción de datos se hacía de manera correcta el siguiente paso fue crear nuestros propios programas en C++ que realizaban la función de envío (*cansend*) y recepción (*candump*).

3.3.1. Librerías utilizadas

Las librerías necesarias para el correcto funcionamiento de la prueba son:

- *iostream*: Cabecera estándar de Entrada/Salida.
- *cstdlib*: Cabecera estándar de propósito general.
- *fstream*: Manejo de directorios y archivos (fclose, fopen,etc.).
- *string*: Manejo de strings.
- *sys/types.h y sys/socket.h:* Librerias para el manejo de sockets.
- *linux/can.h y Linux/can/raw.h*: Librerias para el manejo del bus can en Linux.
- *fcntl.h*: Apertura de ficheros.
- *sys/ioctl.h*: Permite el control y comunicación con drivers.
- *net/if.h*: Nos permite utilizar las estructuras de *if_nameindex.*

3.3.2. Desarrollo del código

Al igual que en el protocolo TCP/IP, en primer lugar, debemos abrir un socket para comunicarnos en la red CAN [6] [7]. La definición de un socket en C es la siguiente:

- *int socket (dominio, tipo, protocolo)*

Para el socket CAN tenemos dos protocolos entre los que elegir:

- Para el modo RAW que es el que emplearemos en el proyecto:
  - *int socket (PF_CAN, SOCK_RAW, CAN_RAW)*
- Para el modo Broadcast:
  - *int socket (PF_CAN, SOCK_DGRAM, CAN_BCM)*

La estructura de una trama CAN, viene ya incluida en la librería *Linux/can.h* y sería como sigue:

```
- struct can_frame {
        canid_t can_id;  /* 32 bit de cabecera*/
        __u8    can_dlc; /* Tamaño de [0..8] */
        __u8    __pad;   /* padding */
        __u8    __res0;  /* reserved / padding */
        __u8    __res1;  /* reserved / padding */
        __u8    data[8] __attribute__((aligned(8)));
  };
```

El socket tiene su propia estructura la cual se define de la siguiente manera:

```
- struct sockaddr_can {
        sa_family_t can_family;
        int can_ifindex;
        union {
            struct {canid_t rx_id, tx_id;} tp;
        } can_addr;
  };
```

Considerando lo anterior, los pasos a seguir para poder trabajar con el bus CAN son:

1. Creamos la estructura

- *int s;*
- *struct sockaddr_can addr;*
- *struct ifreq ifr;*

2. Enlazamos la tarjeta can0 con el socket de la estructura creada anteriormente:

- *strcpy (ifr.if_name, "can0");*

- *ioctl (s, SIOCGIFINDEX, &ifr);*
- *addr.can_family = AF_CAN;*
- *addr.can_ifindex = ifr.ifr_ifindex;*
- *bind (s, (struct sockaddr *) &addr, sizeof(addr));*

Una vez hecha la apertura del socket, podríamos realizar diferentes operaciones sobre él, como:

- *read, wite, send, sento …*

Un aspecto que tener en cuenta en el caso en el que se desee conectar un socket a todas las interfaces CAN, es que el índice de interfaz tiene que ser cero permitiendo al socket recibir tramas CAN desde todas las interfaces habilitadas.

- *can_infindex = ZERO;*

Las operaciones a realizar sobre el socket no podrían ser todas las nombradas anteriormente si no que solo podríamos utilizar:

- *recvfrom, sendto*

3.3.3. Envío trama CAN

El envío de datos a través del bus CAN se lleva a cabo mediante:

- *write (s, &frame, sizeof(struct can_frame));*

En el caso que utilizásemos *can_infindez = ZERO*, se realizaría mediante:

- *sendto (s, &frame, sizeof(frame), 0, (struct sockaddr*)&addr, sizeof(addr));*

Las aplicaciones de intercambio de datos pueden ejecutarse en el mismo nodo o en diferentes nodos sin ningún cambio. Para garantizar que una aplicación reciba la misma información en dos ejemplos diferentes, es necesario un bucle de retorno local de las tramas CAN.

Los dispositivos de red Linux solo pueden manejar transmisión y recepción de cuadros dependientes de los medios. Esto puede producir que la transmisión de una trama CAN con identificador de baja prioridad pueda retrasarse por la recepción de una trama CAN con identificador de alta prioridad debido al arbitraje. Por ello, para conseguir un tráfico de datos correcto en el nodo, el bucle invertido de los datos enviados debe realizarse después de una transmisión exitosa.

La funcionalidad de bucle invertido está habilitada de forma predeterminada mediante la retroalimentación de las tramas, que se realiza de la siguiente manera:

- *int loopback = 0; /* 0 = disabled, 1 = enabled */*
- *setsockopt (s, SOL_CAN_RAW, CAN_RAW_LOOPACK, &loopback, sizeof(loopback));*

La prueba que hemos realizado (ver *pruebaenvio3.cpp*) tiene como objetivo enviar dos tramas previamente definidas a través del dispositivo can0, de tal manera que estas tramas circulen por el bus y al poner el dispositivo can1 en escucha este sea capaz de recibirlas.

La estructura básica del código consiste en:

- Crear la estructura.
- Abrir el socket en modo raw.
- Comprobar que el socket se ha abierto correctamente. En este caso, si el socket es igual a -1 significa que ha habido un error de apertura, en caso contrario se ha abierto con éxito.
- Enlazamos la tarjeta can0 con el socket creado anteriormente.
- Realizar la retroalimentación de las tramas mediante *loopback*.

Lo siguiente es establecer las características de las tramas a enviar y mediante el *write* enviarlas a través del bus.

	Identificador	**Tamaño (DLC)**	**Datos**
Trama 1	555	5	{1, 2, 3, 4, 5}
Trama 2	3A2	8	{1, 2, 3, 4, 5, 6, 7, 8}

Tabla 6. Características Tramas enviadas pruebaenvio3.cpp.

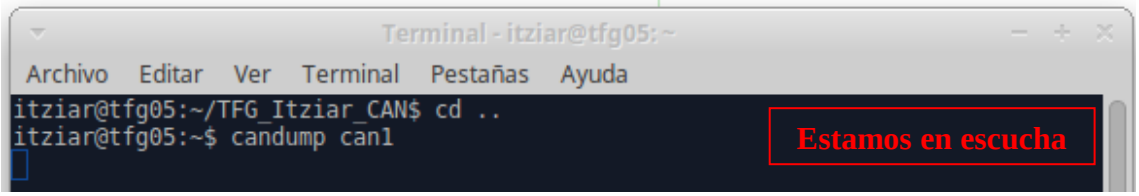

Ilustración 32. Tarjeta can1 en escucha.

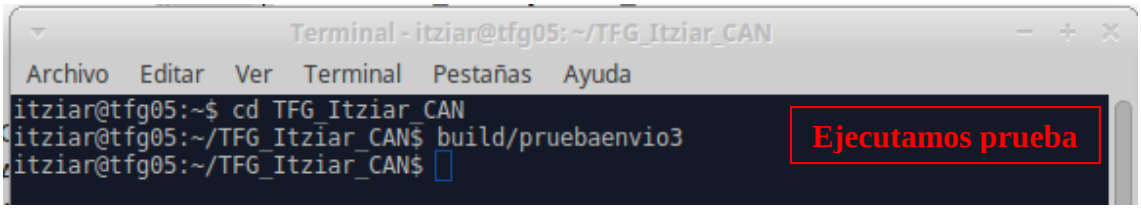

Ilustración 33. Ejecución de la pruebaenvio3.cpp.

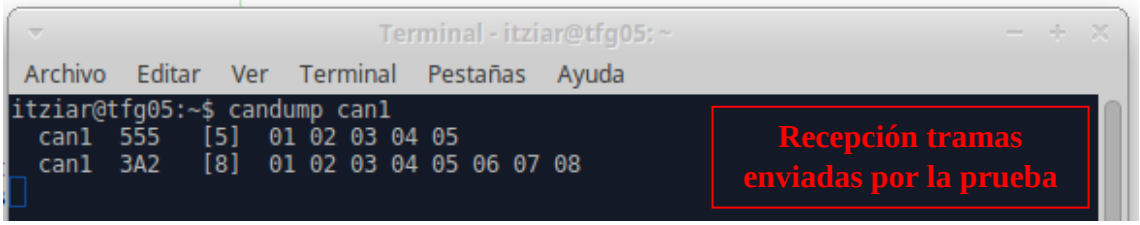

Ilustración 34. Tarjeta can1 recibiendo tramas enviadas por la pruebaenvio3.cpp.

3.3.4. Recepción trama CAN

La recepción de datos a través del bus CAN se lleva a cabo mediante:

- *nbytes = read (s, &frame, sizeof(struct can_frame));*

En el caso que utilizásemos *can_infindez = ZERO*, se realizaría mediante:

- *recvfrom (s, &frame, sizeof(frame), 0, (struct sockaddr*)&addr, sizeof(addr));*

Si *nbytes < 0* → la trama no se leyó o no existe ninguna.
Si *nbytes < sizeof (struct can_frame)* → La trama está incompleta.

El desarrollo de la prueba de recepción de datos (ver *pruebarecepcion.cpp*) tiene como objetivo ser capaz de recibir en la tarjeta can0 aquellas tramas que circulan por el bus y que han sido enviadas mediante la tarjeta can1.

La estructura básica del código consiste en:

- Crear la estructura.
- Abrir el socket en modo raw.
- Comprobar que el socket se ha abierto correctamente. En este caso si el socket es igual a -1 significa que ha habido un error de apertura, en caso contrario se ha abierto con éxito.
- Enlazamos la tarjeta can0 con el socket creado anteriormente.
- Realizar la retroalimentación de las tramas.
- Llevar a cabo un chequeo paranoico.

Un chequeo paranoico se basa en hacer un bucle infinito que escuche continuamente lo que se está enviando por el bus de tal manera que cuando recibe algo sea capaz de detectarlo y mostrarlo. Para ello, trabajamos con el *nbytes* de tal manera que cuando *nbytes = 0* se procederá a la lectura del socket mediante el comando *read*, este comando es bloqueante, es decir, produce que el programa se pare hasta que haga una recepción o se produzca una interrupción o error (*nbytes < 0*). Cuando recibe una trama nos muestra por pantalla el identificador, el tamaño y los datos que han sido recibidos por la tarjeta can0. Si *nbytes* es menor que 0 o menor al tamaño de la estructura, esto significará que la trama está vacía o incompleta, tal y como se muestra en la imagen siguiente:

```
/* paranoid check ... */

while(1){
  nbytes = 0;

  while (nbytes == 0){
    nbytes = read(s, &frame, sizeof(struct can_frame));
  }

  if(nbytes < 0 || nbytes < sizeof(struct can_frame)){
    std::cerr << "La trama esta vacia o incompleta" << std::endl;
    exit(0);
  }

  else{
    std::cout << "Numero de bytes leidos: " << nbytes << std::endl;

    std::cout << "El numero de bytes del paquete es: '" << (int)frame.can_dlc
      << "'" << std::endl;

    std::cout << "El identificador es: " << std::hex << (int)frame.can_id
    << std::endl;

    for(int i = 0; i < frame.can_dlc; i++){
      std::cout << "Dato [" << i << "] = " << std::hex << (int)frame.data[i]
        << std::endl;
    }
  }
}
```

Ilustración 35. Chequeo paranoico.

Las siguientes imágenes nos muestran el caso en el que ejecutamos nuestro programa y se queda a la espera de recibir una trama, a continuación, enviamos varias tramas por la tarjeta can1 y estas son recibidas en nuestro programa:

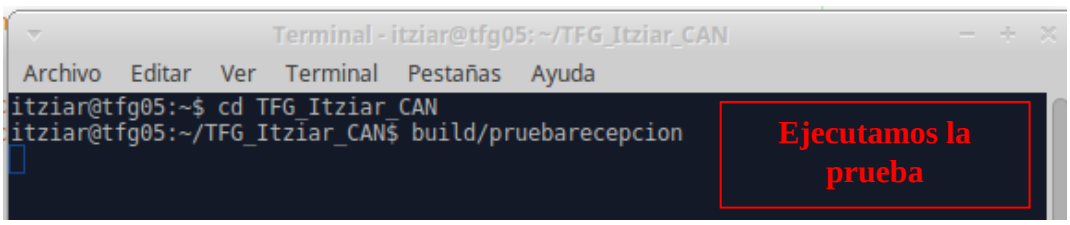

Ilustración 36. Ejecutamos la prueba de recepción de tramas CAN.

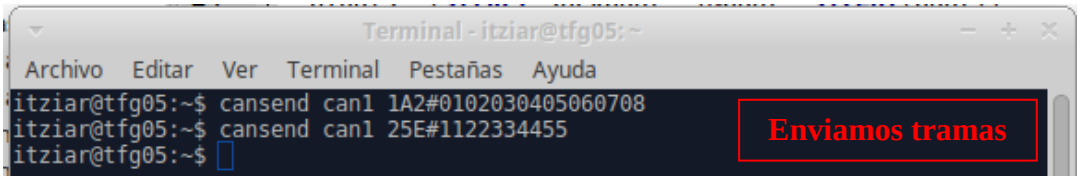

Ilustración 37. Envío de datos por el bus a través de la tarjeta can1.

Ilustración 38. Recepción de las tramas enviadas por la tarjeta can1.

3.4. Creación de clases para el manejo de CAN bus en GNU/Linux

Debido a que las funciones para el manejo de sockets utilizadas se llevan a cabo mediante C, usando su sintaxis y tipo de datos (arrays de bytes, arrays de caracteres), queremos crear unas clases en C++ que oculte esos detalles y nos permita utilizar tipos de datos más cómodos de C++ (vectores y strings).

Para ello, debemos realizar la creación de unas clases en C++ que nos permitan gestionar de manera más cómoda el envío y la recepción de tramas CAN a través del bus, y una prueba para comprobar el correcto funcionamiento de las clases.

La realización de estas clases nos facilitará posteriormente el trabajo, cuando pasemos a establecer la comunicación entre BeagleBone Black y PC.

3.4.1. Clase Trama

La clase Trama (ver *Trama.hpp* y *Trama.cpp*) es un envoltorio (*wrappper*) a la estructura '*can_frame'*. Facilitamos su manejo accediendo a su contenido mediante métodos que soportan diferentes tipos.

Una trama CAN viene dada por su identificador, por su tamaño y por los datos que transfiere. El objetivo de esta clase es obtener o enviar cada uno de estos parámetros en función de si estamos enviando tramas a través del bus o recibiéndolas.

Para poder desarrollar lo anteriormente mencionado la clase trama cuenta con los siguientes atributos:

- *struct can_frame _frame;* → crea la trama CAN, ya que es el que nos da acceso a los parámetros que componen la trama.

Por otra parte, los métodos que hemos empleado para el desarrollo de la clase son:

- *Trama ( );* → constructor de la clase, se encarga de construir la trama con la que vamos a trabajar.
- *unsigned int getId ( );* → obtiene el identificador de la trama.
- *unsigned char getTam ( );* → es el encargado de darnos el tamaño de la trama.
- *void setId (unsigned int id);* → envía el identificador de la trama.
- *void setTam (unsigned char tam);* → se encarga de enviar el tamaño de la trama.
- *unsigned char getDato (int i);* → obtiene un dato de la trama.
- *std::vector <byte> getDatos ( );* → su objetivo es obtener los datos que forman la trama.
- *void setDato (int i, byte valor);* → envía un dato de la trama.
- *void setDatos (std::vector <byte> valores);* → envía los datos que forman la trama CAN.

Con estos atributos y métodos podemos enviar y obtener todos los parámetros que describen a cualquier trama que circule por el bus, ya que tenemos acceso al contenido del *struct can_frame*, gracias a la librería Linux/can.h.

```
//Itziar Rodriguez Hernandez
//Declaración clase trama

#include "Trama.hpp"

Trama::Trama(){

  _frame.can_id = 0;
  _frame.can_dlc = 0;

}

//Nos devuelve el identificador de la trama
unsigned int Trama::getId(){

  return _frame.can_id;
}

//Nos devuelve el tamaño de la trama
unsigned char Trama::getTam(){

  return _frame.can_dlc;
}

//Envia el identificador de la trama
void Trama::setId(unsigned int id){

  _frame.can_id = id;
}

//Envia el tamaño de la trama
void Trama::setTam(unsigned char tam){

  _frame.can_dlc = tam;
}
```

1

Constructor

```
//Nos devuelve el dato recibido
unsigned char Trama::getDato(int i){

  return _frame.data[i];
}

//Nos devuelve los datos recibidos
std::vector <byte> Trama::getDatos(){

  std::vector<byte> datRecibidos;

  for(byte i = 0; i < _frame.can_dlc; i++){

    datRecibidos.push_back(_frame.data[i]);
  }

  return datRecibidos;
}

//Envia un dato
void Trama::setDato(int i, byte valor){

  _frame.data[i] = valor;
}

//Envia los datos
void Trama::setDatos(std::vector <byte> valores){

  //std::cout << "\nLos bytes son: [ ";
  _frame.can_dlc = valores.size();

  for(std::size_t i = 0; i < valores.size(); i++){

    _frame.data[i] = valores[i];
    //std::cout << (int) valores[i] << " ";
  }
    //std::cout << "]" << std::endl;
}
```

2

Ilustración 39. Clase Trama.cpp.

Ahora bien, para poder enviar y recibir tramas enteras a través del bus es necesario crear una clase nueva que nos permita realizar esta función, además, esta clase nueva debe ser amiga de la clase trama para así poder compartir sus métodos y atributos y trabajar de manera conjunta.

3.4.2. Clase GestionCan

La clase GestionCan tiene como objetivo abrir y cerrar el acceso al bus desde nuestro programa, así como, gestionar el envío y recepción de tramas a través del bus CAN (ver *GestionCan.hpp* y *GestionCan.cpp*).

Lo primero que debemos hacer para poder llevar a cabo el objetivo es establecer la amistad entre clases. Para ello, en la zona protegida de la clase Trama debemos indicar que la clase GestionCan es su amiga de la siguiente forma:

- *friend class GestionCan;*

Una vez hecho esto, antes de empezar a desarrollar la clase GestionCan debemos indicar en el *.hpp* que es amiga de la clase Trama añadiendo:

- *class Trama;*

Cuando ya hayamos creado la relación de amistad entre clases podemos pasar al desarrollo de la clase GestionCan.

Para ello, los atributos que necesitaremos serán:

- *bool _inicio;* → indica si el socket está inicializado.
- *int _socket;* → variable entera que hace referencia al socket.

Los métodos necesarios son:

- *GestionCan ( );* → constructor de la clase, tiene como objetivo inicializar los atributos de la clase para utilizarlos posteriormente.

```
#define INVALID_SOCKET -1

GestionCan::GestionCan(){

  _inicio = false;
  _socket = INVALID_SOCKET;
}
```

Ilustración 40. Constructor de la clase GestionCan.

- *bool open (const char* nomInterfaz);* → apertura del socket, es el encargado de crear el socket, comprobar que se ha abierto correctamente, y enlazar la tarjeta seleccionando el interfaz can.

```
bool GestionCan::open(const char *nomInterfaz){

  _inicio = true;

  //Creamos el socket
  _socket = socket( PF_CAN, SOCK_RAW, CAN_RAW );

  if(_socket == -1) {
    std::cerr << "Socketcan: Error de apertura de socket" << std::endl;
    return _socket;
  }

  if(_inicio != true) {
    std::cerr << "No se ha abierto el socket" << std::endl;
    return _socket;
  }

  //Seleccionamos el socket can
  struct sockaddr_can addr;
  //Localizamos la interfaz
  struct ifreq ifr;

  //Obtenemos el indice para can0
  strcpy(ifr.ifr_name, nomInterfaz);
  ioctl(_socket, SIOCGIFINDEX, &ifr);

  //Seleccionamos el interfaz CAN
  addr.can_family = AF_CAN;
  addr.can_ifindex = ifr.ifr_ifindex;

  bind(_socket, (struct sockaddr *)&addr, sizeof(addr));

  return _socket;
}
```

Ilustración 41. Método open clase GestionCan.cpp.

- *bool close ( );* → cierre del socket, su función es cerrar el socket cuando se termine de utilizar.

```
//Cerramos el socket
bool GestionCan::close(){

  if(!_socket){
    return false;
  }
  return true;
}
```

Ilustración 42. Método close clase GestionCan.cpp.

- *bool recv (Trama &tr);* → recibe una trama, es el encargado de recibir las tramas que circulan por el bus y comprobar si es correcta o por el contrario está vacío o incompleta. En caso de ser una trama correcta obtiene los parámetros característicos de la trama.

```
// Recibir un mensaje CAN
bool GestionCan::recv(Trama &tr){

  int tam = sizeof(struct can_frame);

  int nbytes = 0;

  while (nbytes < tam ){

    /*std::cout << "Vamos a intentar leer " << nbytes << std::endl;*/

    int n2 = read(_socket, &(tr._frame)+nbytes, tam - nbytes);

    if(n2 < 0){
      std::cerr << "La trama esta vacia o incompleta" << std::endl;
      return false;
    }

    nbytes = n2 + nbytes;
  }

  /*std::cout << "Numero de bytes leidos: " << nbytes << std::endl;

  std::cout << "El numero de bytes del paquete es: '"
    << (int) (tr._frame).can_dlc << "'" << std::endl;

  std::cout << "El identificador es: " << std::hex
    << (int) (tr._frame).can_id << std::endl;

  for(int i = 0; i < (tr._frame).can_dlc; i++){
    std::cout << "Dato [" << i << "] = " << std::hex
      << (int) tr._frame.data[i] << std::endl;
  }*/

  return true;
}
```

Mensajes de depuración

Ilustración 43. Método recv clase GestionCan.cpp

- *bool send (Trama tr);* → envía una trama, procesando los parámetros de la trama a enviar y mediante el comando *write* realiza el envío.

```
// Enviar un mensaje en Can
bool GestionCan::send(Trama tr){

  (tr._frame).can_id = tr.getId();
  (tr._frame).can_dlc = tr.getTam();

  std::vector <byte> datEnviados = tr.getDatos();

  /*std::cout << "El Identificador del mensaje es: " << std::hex
    << (int) (tr._frame).can_id << std::endl;

  /std::cout << "El numero de bytes del paquete es: '"
    << (int)(tr._frame).can_dlc << " '" << std::endl;*/


  write(_socket, &(tr._frame), sizeof(struct can_frame));

  return true;
}
```

Mensajes de depuración

Ilustración 44. Método send clase GestionCan.cpp

3.4.3. Prueba

La prueba realizada (ver *pruebatrama1.cpp*) tiene como propósito enviar una trama CAN predefinida a través del bus, la cual es recibida por la tarjeta can1 mediante un *candump*, y, posteriormente quedarse en espera de recepción de una nueva trama que se envía por medio de la tarjeta can1 mediante *cansend*.

Esta prueba, a diferencia de la *pruebaenvio3.cpp* y *pruebarecepcion.cpp*, no tiene la necesidad de trabajar con el socket ya que al asociarlas con las clases anteriores serán estás las encargadas de llevar a cabo esos pasos. Por otro lado, la diferencia principal con las pruebas anteriores es que, en este caso, el envío y recepción se hace todo en la misma prueba.

La *pruebatrama1.cpp*, en primer lugar, realiza el envío de una trama CAN que es recibida por la tarjeta can1. Para ello, lo primero que hace es establecer la tarjeta con la que va a trabajar (en este caso can0) y posteriormente abrir el socket. Una vez hecho esto, crea una trama t1 para la cual se establece el identificador y los datos a enviar, y se realiza el envío.

```
//Itziar Rodriguez Hernandez
//Prueba trama 1

#define TITULO "Enviar y recibir datos"

#include "Trama.hpp"
#include "GestionCan.hpp"

int main(){
  std::cout << "\n***  " << TITULO << "  ***\n" << std::endl;

  GestionCan can0;
  can0.open("can0");

  Trama t1;
  t1.setId(0x555);
  std::vector <byte> mensaje1 {1, 2, 3, 4};
  t1.setDatos(mensaje1);
  can0.send(t1);

  Trama t2;
  std::cout << "Pasamos a recibir" << std::endl;

  if(can0.recv(t2)){

    std::cout << "Se ha recibido trama con id " << t2.getId()
      << " y con " << (int) t2.getTam() << " bytes, que son [";

      std::vector<byte> datRecibidos = t2.getDatos();

      for(std::size_t i = 0; i < datRecibidos.size(); i++)
        std::cout << (int) datRecibidos[i] << " ,";
      std::cout << "]" << std::endl;
  }else{

    std::cerr << "Se ha producido un error" << std::endl;
  }
}
```

Abrimos el acceso al bus

Enviamos trama 1

Recibimos trama

Ilustración 45. Pruebatrama1.cpp.

Como vemos en la imagen anterior, gracias a la creación de la clase *Trama* y la clase *GestionCan* el trabajo a realizar es mínimo. Lo único que debemos hacer en nuestra prueba es utilizar los métodos ya creados para la realización de la apertura del acceso al bus y para el envío y recepción de las tramas, y establecer el identificador y los datos de las tramas que queremos enviar. Con esto se muestra la utilidad de trabajar con clases y uno de los objetivos de este proyecto, llevar a cabo un conjunto de clases que nos permitan el manejo del bus CAN de manera cómoda y fácil para desarrollar diferentes pruebas para el envío y recepción de tramas.

Como apreciamos en la imagen, en primer lugar, se envía la trama t1, cuyas características son:

- Identificador: *555*
- Tamaño: *4*
- Datos: *[1, 2, 3, 4]*

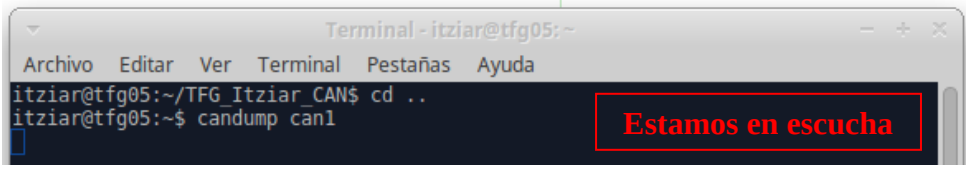

Ilustración 46. Tarjeta can1 en escucha.

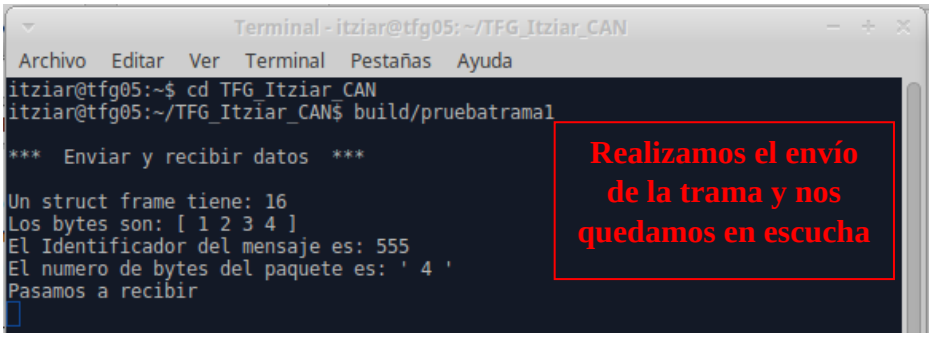

Ilustración 47. Nuestra prueba realiza el envío y se queda en escucha.

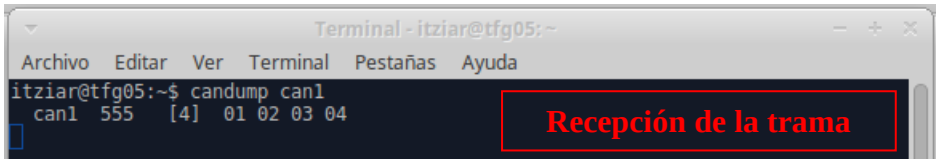

Ilustración 48. Recepción de la trama enviada por la prueba.

Después de enviar una trama, nuestra prueba se queda en escucha esperando la recepción de una trama que se envíe por el bus. Mediante el dispositivo can1 enviamos una nueva trama y vemos como nuestra prueba es capaz de recibir dicha trama y obtener los parámetros que la caracterizan.

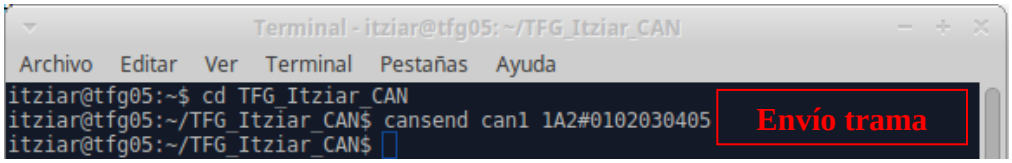

Ilustración 49. Tarjeta can1 envía trama.

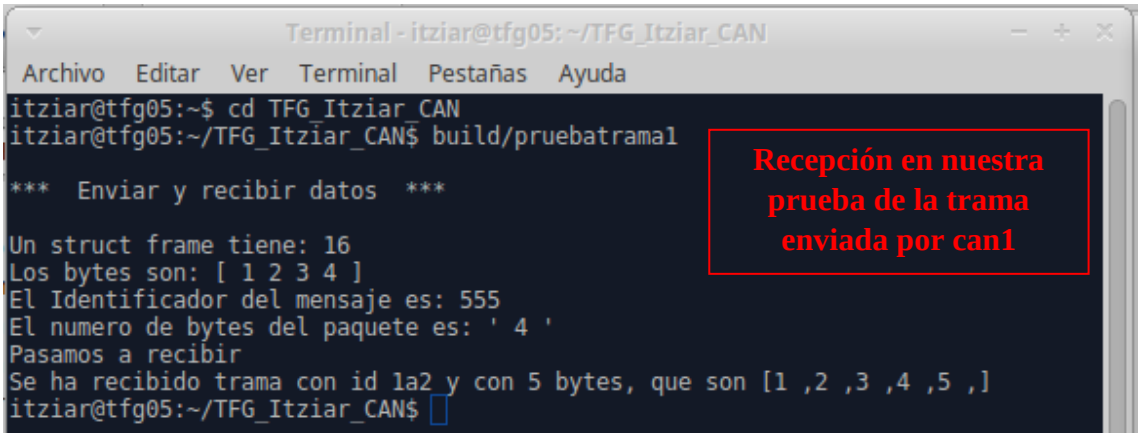

Ilustración 50. Recepción de la trama enviada por la tarjeta can1.

3.5. Creación de clases para controlar entrada analógica de la BeagleBone

La BeagleBone Black consta de 7 entradas analógicas, una entrada de alimentación (tensión de 0 a 1’8V) y una entrada de tierra. Estas entradas nos permiten leer datos analógicos y hacer una transformación analógica/digital para poder trabajar con estos valores en la BeagleBone o en el propio PC.

En nuestro caso, los valores analógicos que vamos a emplear serán los dados por un potenciómetro de un 1 KΩ a partir del cual iremos obteniendo diversos valores de voltaje comprendidos entre 0 y 1’8 V.

El circuito que hemos realizado es el siguiente:

Ilustración 51. Circuito entradas analógicas BBB.

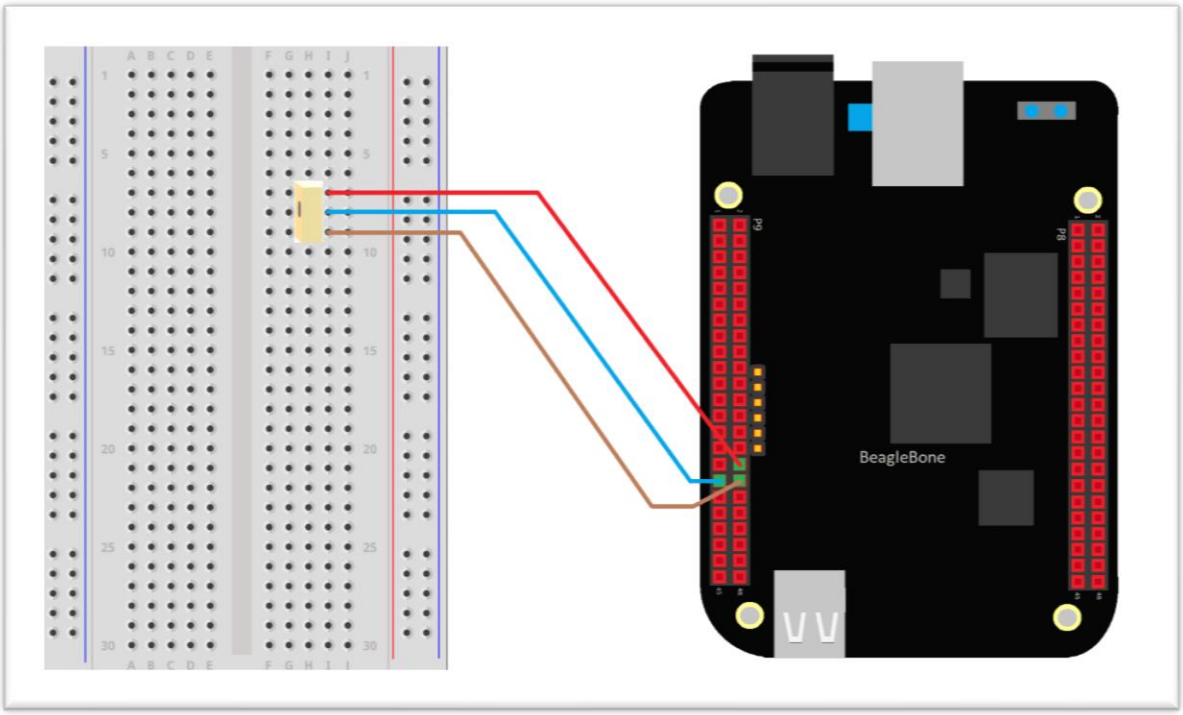

Ilustración 52. Esquema circuito entradas analógicas BBB.

3.5.1. Clase EntradaAnalogica

La clase EntradaAnalógica (ver *EntradaAnalogica.hpp* y *EntradaAnalogica.cpp* dentro de la carpeta BBB) tiene como objetivo realizar la configuración del pin de entrada analógica de la BeagleBone Black con el que vamos a trabajar, acceder a él, leer el valor de voltaje que se está introduciendo y realizar la conversión para ir obteniendo los diferentes valores de tensión suministrados por el potenciómetro.

Para el desarrollo de esta clase los atributos que hemos creado han sido:

- *std::string _ainPath;* → guarda el acceso al archivo del path AINX como un string.
- *int _ainName;* → guarda el número de la entrada analógica con la que vamos a trabajar.

Los métodos necesarios para llevar a cabo el objetivo son los siguientes:

- *EntradaAnalogica (int _ainName);* → constructor de la clase, comprueba que la entrada analógica elegida corresponde con una de las existentes, una vez hecho esto realiza la configuración de dicha entrada para poder ir obteniendo los valores analógicos necesarios.
- *int getValor ( );* → lee el archivo correspondiente a la entrada analógica con la que estamos trabajando y almacena dicho valor en una variable entera.
- *double voltaje ( );* → obtiene los valores de voltaje que está dando el potenciómetro. Para ello, coge lo valores leídos por el método anterior los multiplica por 1'8 (valor de tensión máxima permitida) y los divide entre 4096 que corresponde con el número máximo que nos puede devolver el conversor puesto que este trabaja a 12 bits. De esta manera se obtiene el valor de voltaje que se ha introducido en la entrada analógica.

Como hemos visto en el apartado 2.3 para poder trabajar con los diferentes módulos de la BBB debemos realizar nuestro propio código para el acceso a través del *sysfs*.

Para el módulo ADC existen 7 entradas analógicas en la BBB (AIN0 a AIN6), además de los voltajes de referencia del conversor VDD_ADC y GNDA_ADC.

Para poder conocer el valor de entrada es necesario acceder a los ficheros */sys/bus/iio/devices/iio:device0/in_voltage#_raw* donde # puede ir de 0 a 6. Como el conversor es de 12 bits nos devolverá una cadena de caracteres con un número entre 0 y 4095.

El constructor de nuestra clase será el encargado de permitirnos acceder a estos ficheros y configurar el pin que vayamos a utilizar:

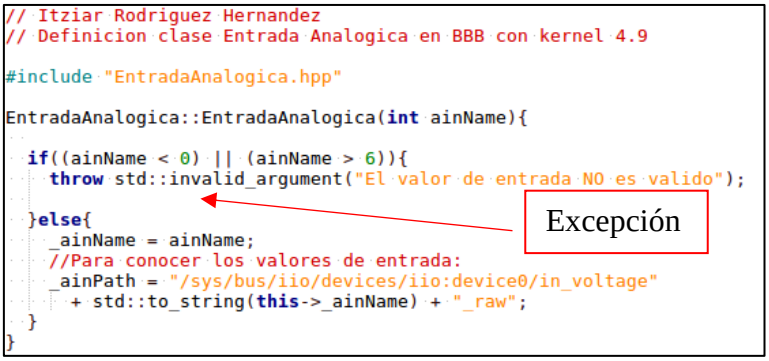

Ilustración 53. Constructor clase, configuración pin analógico.

3.5.2. Prueba

Antes de llevar a cabo la prueba, es necesario realizar la apertura del terminal de la BeagleBone Black, este paso se hará de aquí en adelante siempre que queramos utilizar la BBB.

Lo que debemos hacer es abrir el terminal del PC y enviar el siguiente comando que es el que nos permite convertir ese terminal en un terminal para la BeagleBone Black:

- *ssh debian@192.168.72*

Hecho esto, ya podemos ejecutar nuestra prueba de la misma manera que lo hacíamos en el PC.

Una vez hemos creado la clase necesaria para el control de la entrada analógica de la BeagleBone, podemos llevar a cabo una prueba (ver *pruebaEntradaAnalogica.cpp* dentro de la carpeta BBB) que nos muestre que dicha clase funciona correctamente y que realiza las operaciones que se esperan de ella.

Se ha desarrollado una prueba cuyo funcionamiento básico es muy sencillo. En primer lugar, debemos pasarle a la clase que entrada analógica es la que vamos a utilizar (en nuestro caso es AIN4) y luego iremos obteniendo y mostrando por pantalla los valores de tensión que se están introduciendo por dicha entrada, tomando valores cada 1000 milisegundos (1 segundo).

```
//Itziar Rodriguez Hernandez
//Prueba para probar la entrada analogia de la BBB

#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <thread>

#include "EntradaAnalogica.hpp"

int main(){

  EntradaAnalogica entrada4(4);

  while(true){
    std::cout << "El valor de la entrada 4 es: " << entrada4.voltaje()
      << std::endl;

    std::this_thread::sleep_for(std::chrono::milliseconds(1000));

  }

}
```

Indicamos que pin vamos a usar

Leemos valor analógico

Ilustración 54. PruebaEntradaAnalogica.cpp.

Gracias a la creación de las clases la prueba a realizar es muy sencilla, lo único que debemos hacer es crear un bucle dónde se vayan leyendo valores de la entrada analógica cada segundo y se vayan mostrando por pantalla.

Como se muestra en la imagen, estos valores irán variando entre 0 y 1’8 a medida que vamos girando el potenciómetro:

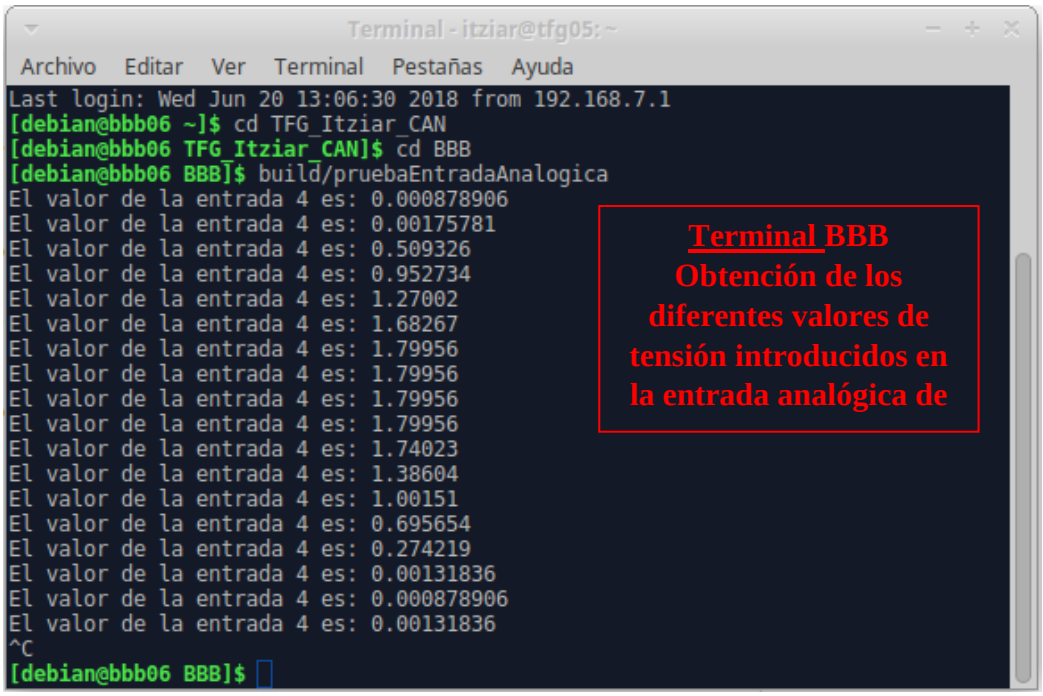

Ilustración 55. Lectura valores de tensión obtenidos de la entrada analógica 4 de la BBB.

3.6. Creación de clases para controlar la señal PWM de la BeagleBone

Una de las características más importantes con las que nos permite trabajar la BeagleBone Black es con el módulo PWM.

La modulación por ancho de pulsos PWM de una señal es una técnica que nos permite modificar el ciclo de trabajo de una señal periódica.

Este ciclo de trabajo corresponde con el ancho relativo a la parte positiva de la señal en relación con el período:

$$D = \frac{\tau}{T} \tag{3.1}$$

Dónde D es el ciclo de trabajo conocido como Duty Cycle, τ es el tiempo en el que la señal se mantiene positiva y T es el período.

Como hemos comentado en el apartado 2.3.2, el módulo PWM de la BBB cuenta con ocho salidas eHRPWM las cuales se han de configurar para unos valores u otros de pwmchip y pwm según la que hayamos elegido como se muestra en la tabla 4 de dicho apartado.

Para visualizar el efecto de la señal PWM desde la BeagleBone hemos realizado un circuito que consta de una resistencia de 330 Ω y un led:

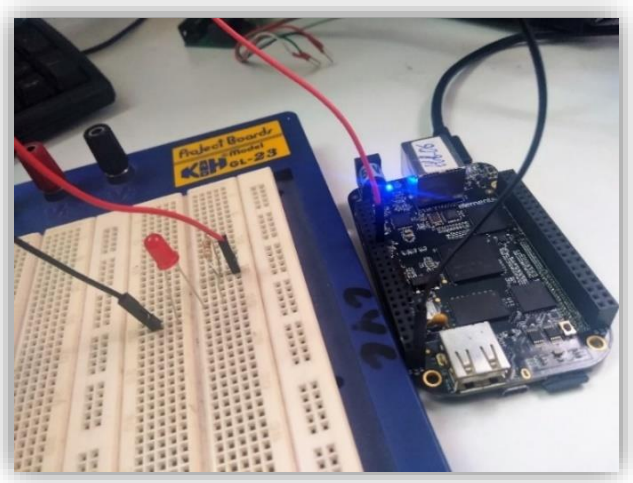

Ilustración 56. Circuito para obtener señales PWM en la BeagleBone Black.

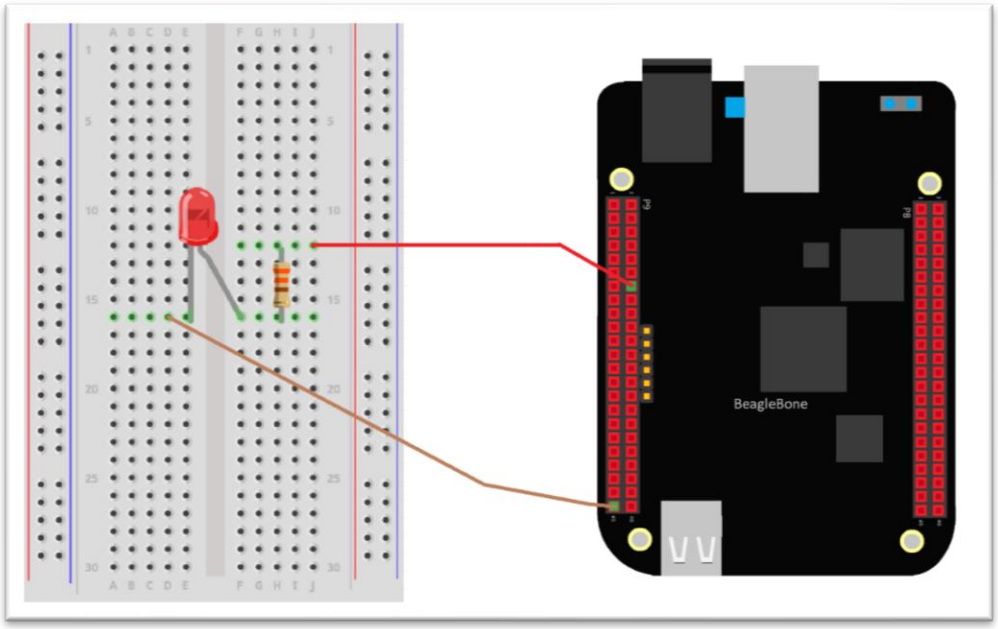

Ilustración 57. Esquema del circuito para obtener señal PWM en la BBB.

El objetivo principal es crear unas clases que nos permitan controlar el período y la variación del porcentaje de duty cycle sobre cualquiera de las salidas PWM. Además, con ayuda del osciloscopio podemos ir viendo las variaciones que se van produciendo sobre la señal a medida que cambia el valor del porcentaje de duty para el periodo establecido.

Para probar el funcionamiento de esas clases, haremos un programa de prueba en el que un bucle ascendente y descendente hará que un LED se vaya encendiendo y apagando progresivamente.

3.6.1. Clase LecYEsc

La clase LecYEsc (ver *LecYEsc.hpp* y *LecYEsc.cpp* dentro de la carpeta BBB) establece las funciones de escritura y lectura de los archivos que contienen los diferentes datos de PWM para posteriormente ser utilizada en la clase hija Pwm permitiendo realizar el envío y recepción de los diferentes parámetros.

Al tratarse de una clase madre y cuya utilidad es únicamente permitir lectura y escritura de archivos, solo cuenta con los siguientes métodos:

- *int write (std::string path, std::string nombreArch, std::string valor);* → escribe en el archivo especificado por el path y el nombre de archivo que se le pasa, el valor que se le está pasando como un string.
- *int write (std::string path, std::string nombreArch, int valor);* → escribe en el archivo especificado por el path y el nombre de archivo que se le pasa, el valor que se le está pasando como entero.
- *std::string read (std::string path, std::string nombreArch);* → lee el contenido del archivo que viene por el path y el nombre de archivo que se le pasa.

3.6.2. Clase Pwm

La clase Pwm (ver *Pwm.hpp* y *Pwm.cpp* dentro de la carpeta BBB) es clase hija de la clase LecYEsc. Su objetivo principal es realizar el control y la obtención de parámetros que describen a la modulación PWM.

Una señal PWM se caracteriza por el período de trabajo y por el ciclo de trabajo, además, también es importante la polaridad de la señal para saber si está activada a alta o a baja. Con estos tres parámetros podemos transmitir una señal con facilidad.

La clase Pwm lo que busca es establecer los parámetros nombrados para poder visualizar la señal de trabajo, así como, obtener estos valores para comprobar el correcto funcionamiento. Además, es la encargada de configurar y exportar los pines y archivos necesarios para poder llevar a cabo los objetivos propuestos.

La clase Pwm está formada por los siguientes atributos:

- *std::string _path;* → el path está compuesto por el siguiente formato: "*/sys/class/pwm/pwmchipX/pwmX/"* dónde X dependerá el pin elegido y viene dado por las variables locales del constructor. Este atributo es el que nos permite leer y escribir el contenido del archivo de trabajo.
- *std::string _nombre;* → en el atributo nombre se almacena el path con el que se va a trabajar. Una vez tenemos el nombre ya podemos determinar el chip y el canal fácilmente según lo establecido en la tabla 4.
- *std::string _sys;* → el sys está formado por "*/sys/class/pwm/pwmchipX/"* y es el atributo necesario para realizar la exportación del path y poder acceder a los distintos archivos que componen la señal PWM.
- *double _periodoNsFrec (unsigned int);* → tiene como objetivo devolver el valor de la frecuencia, en caso de que el periodo haya sido pasado en nanosegundos.
- *unsigned int _frec2PeriodoNs (double);* → se encarga de devolver el periodo en segundos en caso de que se haya pasado la frecuencia de la señal en Hz.

Además, esta clase dispone de las siguientes variables locales del constructor:

- *const static std::vector <std::string> pines;* → tiene el objetivo de guardar todos los pines de la BBB empleados para PWM para determinar el chip y el valor del canal pwm necesarios para realizar la configuración.
- *const static std::vector <int> pin2chip;* → es el encargado de guardar todos los chip posibles a elegir en función del pin que se va a utilizar

- *const static std::vector <int> pin2pwm;* → tiene la misión de guardar el canal pwm que se necesita según el pin con el que vayamos a trabajar.

Una vez hemos establecido los atributos y las variables locales del constructor necesarias para la configuración de la señal PWM, debemos llevar a cabo los métodos que se necesitan:

- *enum POLARITY{*
  ACTIVO_BAJO = 0,
  ACTIVO_ALTO = 1
  }; → Establece la polaridad de la señal en función de si nos interesa activarla a alta o activarla a baja.
- *Pwm (std::string pwmName);*→ constructor de la clase, configura y exporta el path (como se indica a continuación) que vamos a utilizar para poder así trabajar con él. Según el pin que se le pase realizará la configuración de un chip y un canal determinado.
- *void enable (bool habilitar);* → nos permite habilitar el módulo PWM para poder realizar la transmisión de la señal.
- *int setPeriodo (unsigned int periodoNs);* → envía el período en nanosegundos con el que vamos a trabajar.
- *unsigned int getPeriodo ( );* → obtiene el periodo con el que estamos trabajando.
- *int setFrecuencia (double frecuenciaHz);* → envía la frecuencia que se desea utilizar.
- *double getFrecuencia ( );* → obtiene la frecuencia que se está utilizando.
- *int setDutyCycle (unsigned int dutyNs);* → establece el ciclo de trabajo (duty cycle) de la señal en nanosegundos.
- *int setDutyCycleP (double porcentaje);* → similar al anterior, pero en este caso se establece el dut cycle de la señal en forma de porcentaje.
- *unsigned int getDutyCycle ( );* → obtiene el duty cycle de la señal en nanosegundos.
- *double getDutyCycleP ( );* → se encarga de obtener el duty cycle de la señal en forma de porcentaje.
- *int setPolaridad (Pwm::POLARITY);* → determina la polaridad con la que se va a trabajar.
- *void invertirPolaridad ( );* → invierte la polaridad que se está utilizando.
- *Pwm::POLARITY getPolaridad ( );* → determina la polaridad con la que se está trabajando.
- *~Pwm ( );* → destructor de la clase.

La configuración de los pines realizada por el constructor PWM necesaria para el manejo del módulo PWM se hace a través del *sysfs* en el path */sys/class/pwm/pwmchip#* dónde # y la relación de los pines viene establecido según la tabla 4.

En cada una de esas carpetas existe un fichero *export* que da acceso a la configuración del PWM correspondiente. Lo primero es *exportar* el canal deseado.

Con esto aparecerá la carpeta pwm#, dentro de la cual están *period* y *duty_cycle* que expresan tiempos de la PWM (en nanosegundos) y *enable* que poniéndolo a 1 comienza a generarse la PWM y a 0 se desactiva.

A continuación, se debe configurar el *pinmux* para selecciones de la salida PWM ya que los pines pueden tener otras funcionalidades:

- *config-pin P9_16 pwm*

Aunque la exportación se puede usar como usuario *debian*, las carpetas *pwm#* creadas son sólo del *root* por lo que los programas deben ejecutarse como *root* (usando *sudo*).

Además, hemos creado el constructor de la clase Pwm de tal manera que nos facilite el trabajo todo lo posible. El constructor tiene una tabla dónde están los ficheros en función del nombre del pin, el canal, etc., en las pruebas no será necesario indicar esto para manejar una PWM si no que se hará uso de la clase despreocupándonos de realizar cada configuración.

```
Pwm::Pwm(std::string pwmName){
  this->_nombre = pwmName;
  this->_frecuenciaAnalog = 100000;
  this->_vAnalogMax = 3.3;

  const static std::vector <std::string> pines {"P9_42", "P9_22"
    , "P9_21","P9_14", "P9_16", "P9_28", "P8_19", "P8_13" };
  const static std::vector <int> pin2chip { 0, 1, 1, 3, 3, 5, 6, 6 };
  const static std::vector <int> pin2pwm { 0, 0, 1, 0, 1, 0, 0, 1 };

  unsigned int i;

  for(i = 0; i < pines.size(); i++){
    if(_nombre == pines[i])
      break;
  }
  if (i == pines.size())
    throw std::invalid_argument("El valor de entrada NO es valido");

  this->_sys = PWM_PATH + std::to_string(pin2chip[i]) +"/";

  this->_path = PWM_PATH + std::to_string(pin2chip[i]) + "/pwm"
    + std::to_string(pin2pwm[i]) + "/";

  std::string configuracion = "config-pin " + _nombre + " pwm";

  std::system(configuracion.c_str());  //ejecuta comando del sistema

  //exportamos
  std::string exportPath = _sys + "export";
  std::ofstream fExport(exportPath);
  fExport << "1" << std::endl;
  fExport.close();

}
```

Ilustración 58. Constructor PWM, configuración de pines PWM.

3.6.3. Prueba

La prueba (ver *pruebaPWM.cpp* en la carpeta BBB) que se ha desarrollado, en primer lugar, indica el pin con el que se va a trabajar para que así las clases anteriores puedan realizar la configuración y acceder a los archivos correspondientes. El siguiente paso es establecer el periodo, el porcentaje de duty y la polaridad de la señal con la que queremos trabajar y poner el enable a true para poder general la señal.

Después de haber establecido las características de trabajo, para poder ir viendo el comportamiento de la señal lo que debemos hacer es ir variando los valores de porcentaje del duty cycle de manera que el LED se vaya encendiendo cada vez más para luego ir apagándose poco a poco. Esto se verá reflejado en el osciloscopio con una modificación de la señal según los valores de duty que vamos teniendo.

```cpp
//Itziar Rodriguez Hernandez
//Prueba para probar las entradas PWM de la BBB

#include <iostream>
#include <fstream>
#include <string>
#include <chrono>
#include <cstdlib>
#include <thread>

#include "Pwm.hpp"

int main(){
  Pwm entradaPWM("P9_16");

  entradaPWM.setPeriodo(100000);
  entradaPWM.setDutyCycleP(50.0);
  entradaPWM.setPolaridad(Pwm::ACTIVO_BAJO);
  entradaPWM.enable(true);

  std::cout << "PWM Periodo: " << entradaPWM.getPeriodo() << std::endl;

  for(int i = 0; i <= 100; i = i + 10){

    entradaPWM.setDutyCycleP(i);
    std::cout << "PWM DutyCycle Porcentaje: " << entradaPWM.getDutyCycleP()
      << "%" << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  }

  for(int i = 90; i >= 0; i = i - 10){
    entradaPWM.setDutyCycleP(i);
    std::cout << "PWM DutyCycle Porcentaje: " << entradaPWM.getDutyCycleP()
      << "%" << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  }
  return 0;
}
```

Indicamos pin PWM

Establecemos periodo y duty

Aplicamos diferentes porcentajes de duty

Ilustración 59. PruebaPWM.cpp.

La prueba para el control de la señal PWM se realiza de manera cómoda gracias a la creación de las clases *Pwm* y *LecYEsc*, que nos permiten que únicamente pasándole el nombre del pin a trabajar estas realicen toda la configuración y exportación necesaria, y, además, sean las encargadas de fijar y obtener los valores de los diferentes parámetros.

En la siguiente imagen se demuestra lo que hemos explicado con mayor claridad:

```
[debian@bbb06 BBB]$ sudo build/pruebaPWM
[sudo] password for debian:
PWM Periodo: 100000
PWM DutyCycle Porcentaje: 0%
PWM DutyCycle Porcentaje: 10%
PWM DutyCycle Porcentaje: 20%
PWM DutyCycle Porcentaje: 30%
PWM DutyCycle Porcentaje: 40%
PWM DutyCycle Porcentaje: 50%
PWM DutyCycle Porcentaje: 60%
PWM DutyCycle Porcentaje: 70%
PWM DutyCycle Porcentaje: 80%
PWM DutyCycle Porcentaje: 90%
PWM DutyCycle Porcentaje: 100%
PWM DutyCycle Porcentaje: 90%
PWM DutyCycle Porcentaje: 80%
PWM DutyCycle Porcentaje: 70%
PWM DutyCycle Porcentaje: 60%
PWM DutyCycle Porcentaje: 50%
PWM DutyCycle Porcentaje: 40%
PWM DutyCycle Porcentaje: 30%
PWM DutyCycle Porcentaje: 20%
PWM DutyCycle Porcentaje: 10%
PWM DutyCycle Porcentaje: 0%
[debian@bbb06 BBB]$
```

Terminal BBB
Muestra de cómo se va modificando el porcentaje de duty cycle de la señal PWM

Ilustración 60. Variación del porcentaje de duty cycle de la señal PWM.

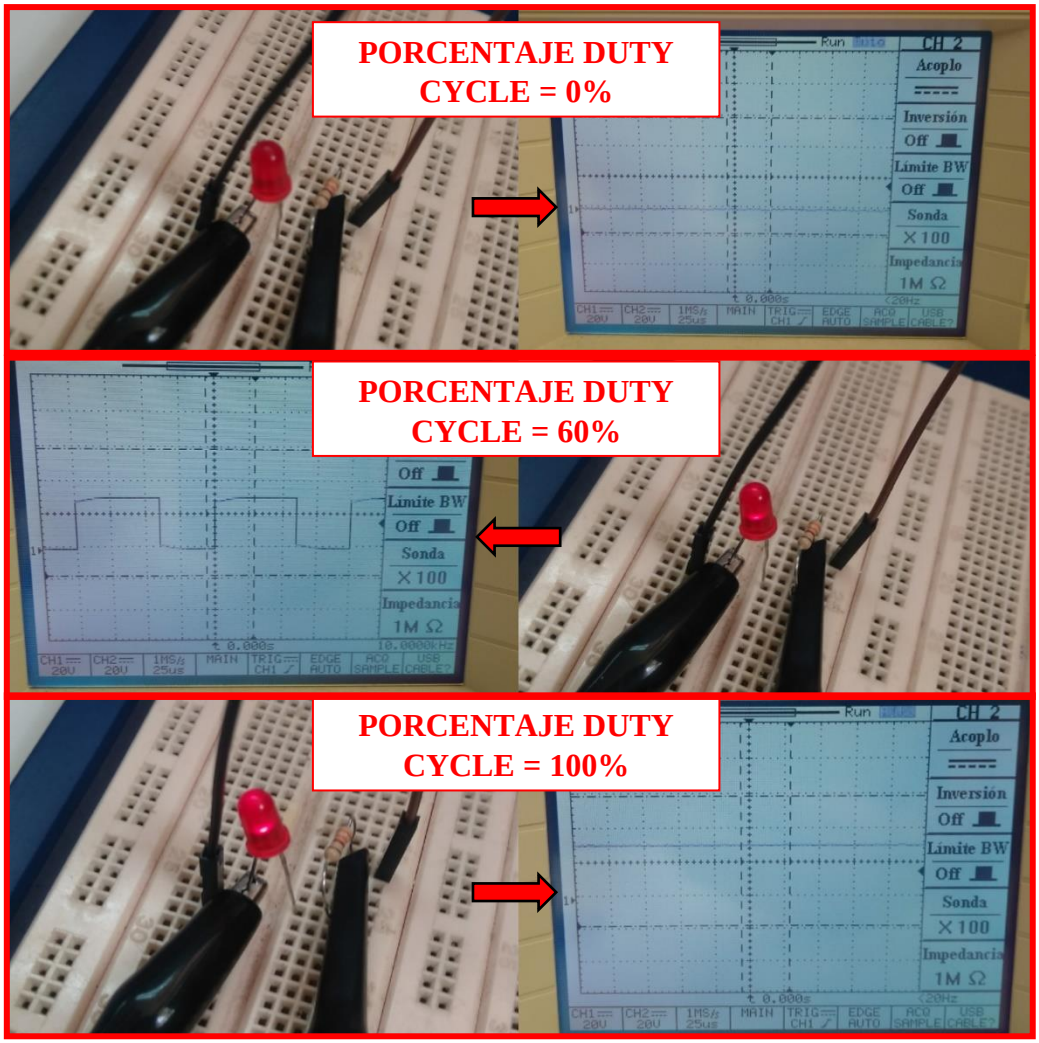

Ilustración 61. Visualización del comportamiento de la señal y el circuito al ir variando el duty cycle.

3.7. Comunicación PC-BBB. Envío y recepción de datos.

Una vez hemos creado las clases para trabajar en el PC y en la BBB de manera separada, a continuación, podemos utilizar estas clases para crear varias pruebas que nos permitan, por medio del bus CAN, realizar una comunicación entre ambos dispositivos.

Lo primero que debemos hacer es utilizar el adaptador basado en SN65HVD230 para poder comunicar la BBB al cable del bus CAN. Debemos tener en cuenta que, mientras que el PC tiene disponibles ambas tarjetas CAN (can0 y can1), la BeagleBone Black solo podrá utilizar una, en nuestro caso hemos elegido can0.

Debemos conectar el adaptador por un lado al cable de CAN bus diferenciando entre el CANH y el CANL, y, por el otro lado, a la BeagleBone Black como se muestra a continuación:

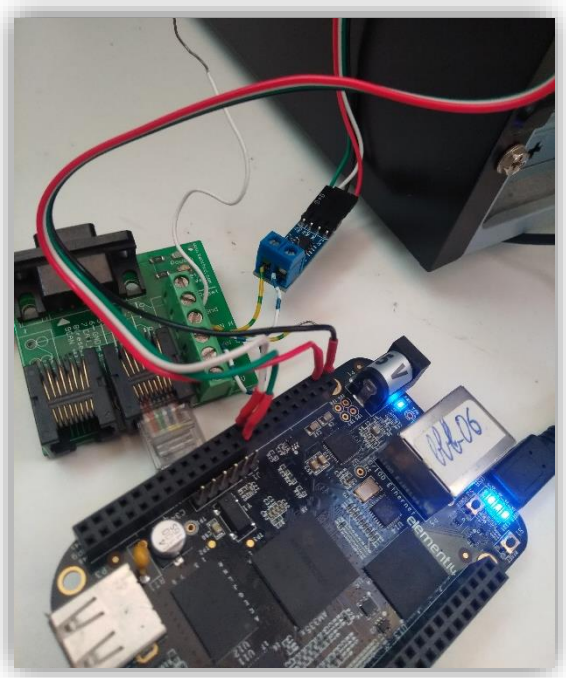

Ilustración 62. Conexión CAN bus BeagleBone Black.

Pin	Función	Conexión
P9_19	DCAN0_RX	CAN RX
P9_20	DCAN0_TX	CAN TX
P9_01	DGND	GND
P9_03	VDD_3V3	3.3 V

Tabla 7. Conexión adaptadores basado en SN65HVD230 a la BeagleBone Black.

Dado que estos pines pueden tener hasta 8 funcionalidades distintas, para poder utilizarlos como bus CAN es necesario configurarlos y levantarlos mediante los siguientes comandos:

- *config-pin P9_19 can* → nos permite configurar este pin como uso CAN.
- *config-pin P9_20 can* → nos permite configurar este pin como uso CAN.
- *sudo ip link set can0 up type can bitrate 125000* → realiza el levantamiento para el uso del pin como tarjeta can0.
- *sudo ifconfig can0 up* → configura el pin que hemos levantado como can0.

Para comprobar la correcta conexión y funcionamiento de la comunicación entre ambos dispositivos hemos creado dos pruebas para el envío y recepción de datos.

La prueba de envío (ver *pruebaenviodatos.cpp*) envía una trama CAN previamente definida desde la BeagleBone Black para que pueda ser recibida en el PC.

También se puede realizar el mismo procedimiento al revés, es decir, podemos enviar la trama desde el PC y recibirla en la BBB.

La trama que se desea enviar tiene las siguientes características:

- Identificador: *1A3*
- Datos: *[1, 2, 3, 4, 5, 6, 7]*

Para poder enviar esta trama por el bus CAN lo primero que debemos hacer es, utilizando las clases ya creadas, abrir la tarjeta can0, crear la trama, establecer el identificador y datos a enviar y realizar el envío.

Una vez hemos hecho esto, podemos pasar a realizar una nueva prueba que nos permita recibir la trama anterior.

La prueba de recepción de tramas (ver *pruebarecibirdatos.cpp*) recibe en el PC la trama que ha sido enviada desde la BeagleBone Black. Esta prueba debe abrir la tarjeta can por la que se está realizando el envío y hacer una recepción de la trama obteniendo el identificador, el tamaño, y el conjunto de datos que se han enviado.

En las imágenes siguientes se muestra lo que se ha explicado, vemos como ejecutando la prueba de recepción desde el terminal del PC esta se queda en espera de que le envíen una trama, mientras que, desde el terminal de la BBB ejecutamos nuestra prueba de envío, comprobando en el terminal del PC que se ha recibido la trama que se espera de manera correcta.

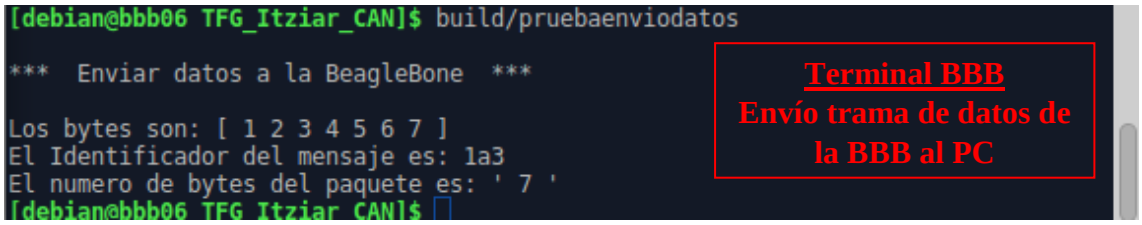

Ilustración 63. Envío de trama de datos de la BBB al PC.

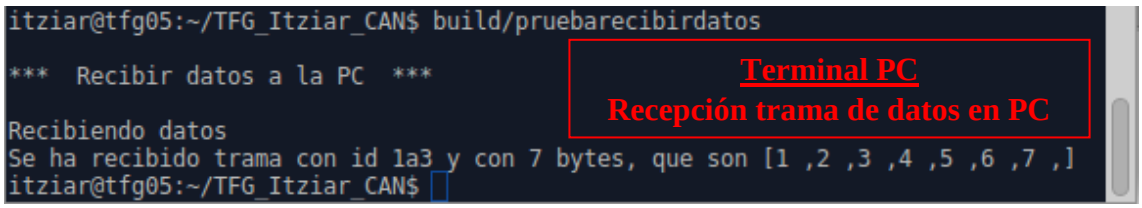

Ilustración 64. Recepción de trama de datos en el PC procedente de la BBB.

3.8. Comunicación PC-BBB. Envío y recepción de datos analógicos

La comunicación entre el PC y la BBB nos permite enviar también otro tipo de datos como pueden ser datos analógicos.

Para poder llevar a cabo dicho envío hemos creado dos pruebas, una que nos permita ir leyendo el valor analógico que se va introduciendo en la BeagleBone y transformarlo en un valor de bytes para crear un paquete de datos con el voltaje correspondiente y enviarlo desde la BBB a través del bus CAN.

Y otra prueba que reciba la trama enviada en el PC, compruebe que es la trama que necesita y obtenga el identificador y los datos recibidos, transformando este paquete de datos en el valor de voltaje que nos está introduciendo el potenciómetro.

De esta manera, la prueba de envío (ver *pruebaAnalogicaEnv.cpp*) lee el valor de la entrada analógica (en nuestro caso la número 4) y va cogiendo este valor y le aplica una máscara que nos permita transformar cada uno de sus dígitos en bytes, obteniendo finalmente 3 bytes los cuales formaran el vector de bytes que corresponde con el paquete de datos a enviar.

En el caso del envío de datos que ocupan 1 byte en paquete CAN, por ejemplo, datos en punto flotante, es necesario tener un convenio definido entre los dispositivos conectados al bus, ya que estos posiblemente tendrán diferentes arquitecturas, representarán los datos '*float*' y '*double*' con distintos tamaños y formatos.

La opción que se ha utilizado es usar la representación en punto fijo con tamaños de bytes concretos a la vista del rango de valores de la variable a transmitir. Este convenio podría dependen del identificador del paquete.

```
std::vector<byte> double2vbyte3(double voltaje) {

  typedef unsigned char byte;

  std::vector<byte> datos;
  int vint = voltaje* 100000;

  byte b0 = vint & 0xff;
  byte b1 = (vint >> 8) & 0xff;
  byte b2 = (vint >> 16) & 0xff;

  datos.push_back(b2);
  datos.push_back(b1);
  datos.push_back(b0);

  return datos;

}
```

Ilustración 65. Transformar el voltaje a un paquete de bytes.

Una vez hecho esto, se abrirá la tarjeta can0, se creará una trama, se asignará un identificador y los datos anteriormente nombrados, para proceder al envío completo de la trama.

La prueba que hemos creado está realizada de tal manera que realiza un envío de trama cada segundo modificando el valor de voltaje y por tanto del paquete de datos mediante el potenciómetro.

Para poder recibir dicha trama, se ha desarrollado una prueba de recepción de datos analógicos (ver *pruebaAnalogicaRec.cpp*) la cual se encarga de comprobar que la trama que circula por el bus corresponde con la trama de datos analógicos y en caso correcto nos muestra el identificador de la trama y el valor de voltaje, pero ya convertido al valor real y no como paquete de datos.

```
double vbyte2int(std::vector<byte> datos) {

  double vreci = ((datos[0] << 16) | (datos[1] << 8) | (datos[0]))/100000;

  return vreci;

}
```

Ilustración 66. Transformar el paquete de bytes a entero.

La prueba de recepción de voltaje lo que hace es establecer la recepción, comprobar que el identificador es el que desea y recibir los datos. Una vez recibe los datos se cogen los bytes que lo forman y se introducen como un *double* para así obtener el voltaje.

Cuando hayamos hecho la transformación, la prueba nos muestra por pantalla el identificador de la trama recibido, el tamaño de la trama, y el valor de voltaje que se está introduciendo por la entrada analógica de la BeagleBone. Cada vez que la BBB realiza un envío este es recibido por la prueba y mostrado por pantalla.

El circuito empleado coincide con el del apartado 3.6.

En las imágenes mostradas a continuación se aprecia con claridad el funcionamiento del envío y recepción de datos analógicos entre PC y BBB:

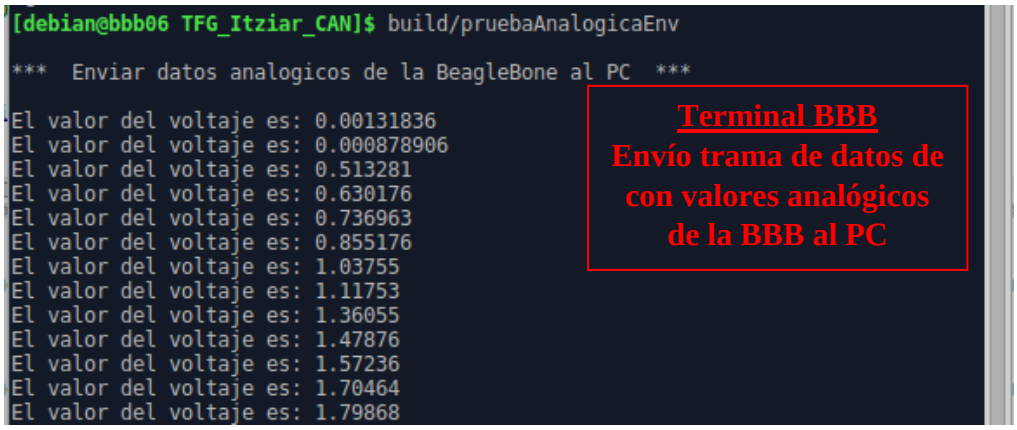

Ilustración 67. Envío de tramas con datos analógicos leídos en la BBB al PC.

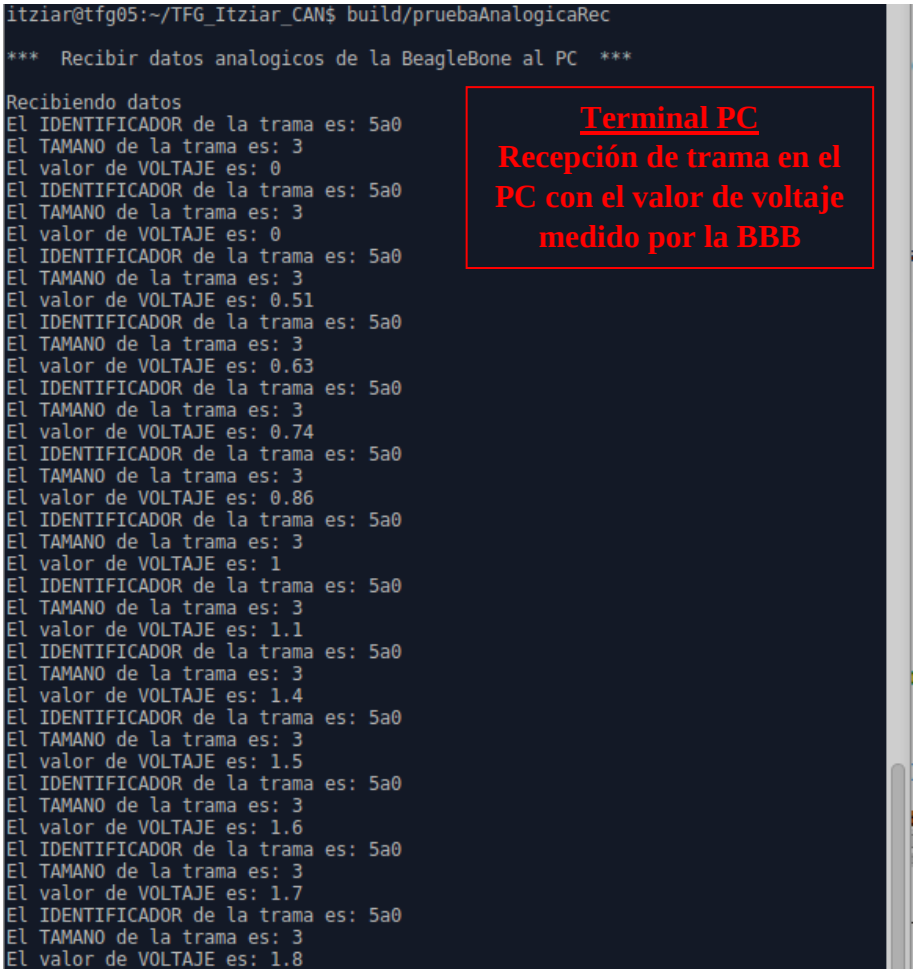

Ilustración 68. Recepción tramas en el PC con valores analógicos leídos por la BBB.

3.9. Comunicación PC-BBB. Envío y recepción de datos PWM.

Otra opción que podemos realizar en la comunicación entre el PC y la BeagleBone Black es el envío y recepción de datos PWM.

Haciendo uso de las clases creadas para el control de una señal PWM en la BeagleBone y utilizando las clases de comunicación de envío y recepción de tramas en el PC es posible crear unas pruebas que nos permitan solicitar los parámetros PWM que se quieren asignar, guardarlos como paquete de datos y enviarlos a través del bus por diferentes tramas, cada una de ellas con un identificador especifico puesto que se trata de datos diferentes.

Para llevar a cabo la comunicación se desarrollan dos pruebas, una que pida al usuario el periodo y porcentaje de duty cycle deseado, enviando este periodo y duty en diferentes tramas a través del bus, y otro, que reciba estas tramas y nos muestre con que parámetros está trabajando el PWM.

Así, el objetivo de la prueba de envío (ver *pruebaPWMEnv.cpp*) es pedir por pantalla desde el terminal de la BeagleBone al usuario que introduzca el valor de duty cycle y período que desea aplicar, y guardar estos valores como bytes. A continuación, creará dos tramas diferentes dónde una se dedicará a la transmisión del período y la otra a la transmisión del porcentaje del duty cycle. Creadas las tramas, el siguiente paso es abrir el bus y realizar el envío para que el PC pueda recibir la información correspondiente.

La prueba de recepción para el PC (ver *pruebaPWMRec.cpp*) abre el bus y obtiene las tramas que circulan por él. Esta prueba está creada de tal manera que es capaz de distinguir por el identificador que tipo de datos contiene la trama y así realizar la transformación para mostrar por pantalla el identificador y tamaño de la trama recibida, así como el tipo de dato que ha recibido y el valor de este.

Las características de cada una de las tramas enviadas son las siguientes:

	Identificador	DLC	Datos
Periodo	123	1	{50}
Porcentaje Duty Cycle	12D	1	{a0}

Tabla 8. Características tramas enviadas a través del bus CAN.

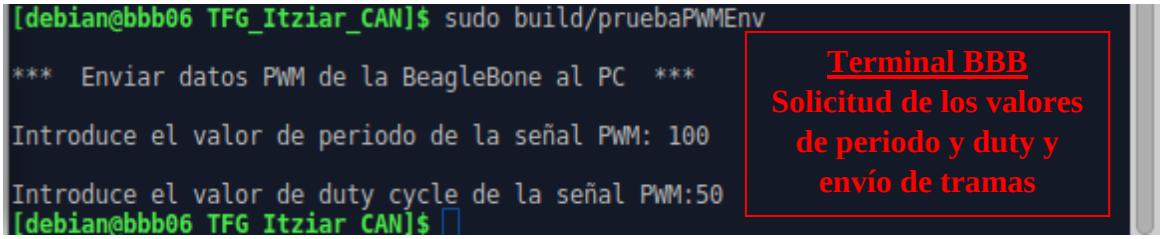

Ilustración 69. Envío datos PWM de la BBB al PC.

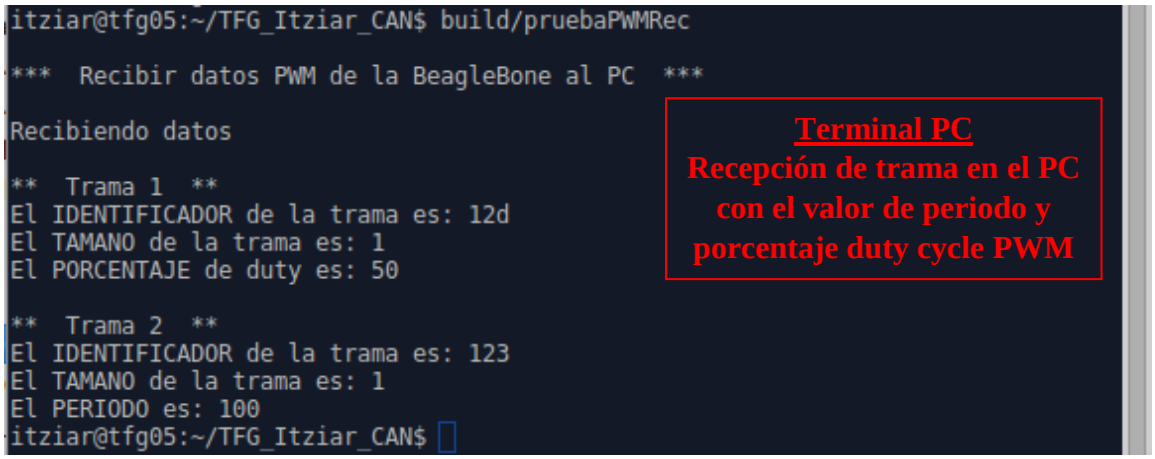

Ilustración 70. Recepción en el PC de los datos PWM enviados por la BBB.

3.10. Sistema de Control Distribuido. Controlador PID para temperatura.

El último paso que vamos a elaborar consiste en llevar a cabo un sistema de control distribuido mediante un controlador PID para el control de un sistema de temperatura simulado mediante un pequeño un circuito. Para ello, se hará uso de todas las clases creadas anteriormente.

Para poder realizar el control PID es necesario el uso del módulo PWM y del módulo ADC de la BBB. La utilización de estos módulos es vital puesto que el circuito en el cual debemos controlar la temperatura (Ilustración 57) tiene un voltaje de entrada que nos permite aumentar la temperatura y que alimentaremos con el módulo PWM para poder controlar con que fuerza realizaremos el calentamiento. Este circuito se encuentra en la placa de experimentos DISEN-EXP que trabaja como máximo a 5V.

En la salida del circuito tenemos una señal analógica que debemos convertirla en una señal digital para que la BeagleBone puede trabajar con ella y saber a qué temperatura se encuentra la resistencia para realizar el lazo de control.

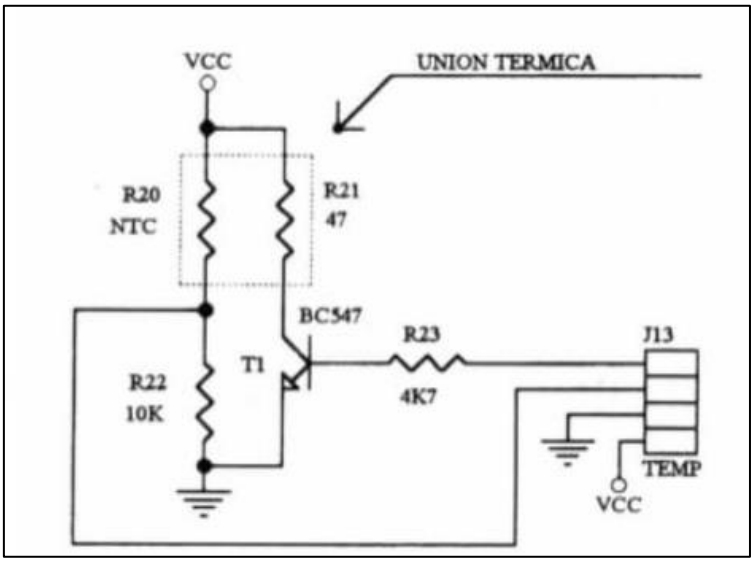

Ilustración 71. Circuito sobre el que se realiza el control PID.

Lo primero que podemos pensar es conectar la salida del circuito directamente a la entrada analógica de la BBB, pero, como hemos visto, la entrada analógica de la BeagleBone solo permite un voltaje de entrada máximo

de 1.8 V. Dado que la salida puede llegar a un máximo de 5 V será necesario reducir este valor de tensión antes de cargarlo en la BBB.

La primera solución que se intentó llevar a cabo fue realizar un partidor de tensión cuyo voltaje de entrada sea 5 V y mediante el juego de las resistencias obtener un valor de salida de 1.8V. Sin embargo, una vez llevamos esto a la práctica nos dimos cuenta de que el voltaje de salida era mucho más pequeño que 1.8 V.

Esto se debe a que el circuito a controlar produce una intensidad tan pequeña que no permite que pase toda la tensión que se desea.

Ante este problema, la solución que tomamos consistió en utilizar un operacional LM358 para amplificar la intensidad que circula y conseguir que en la salida del partidor de tensión saliera como máximo 1.8 V.

También fue necesario añadir un convertidor de nivel para aumentar el valor de tensión dado por la PWM de 3.3 V a 5 V y que el PID se aplicará de manera más rápida y eficaz.

El circuito final es el que se muestra a continuación, dónde el circuito a controlar es alimentado mediante la señal PWM de la BBB, la salida de este la conectamos directamente al amplificador, y la salida del partidor de tensión alimenta a la entrada analógica de la BeagleBone Black.

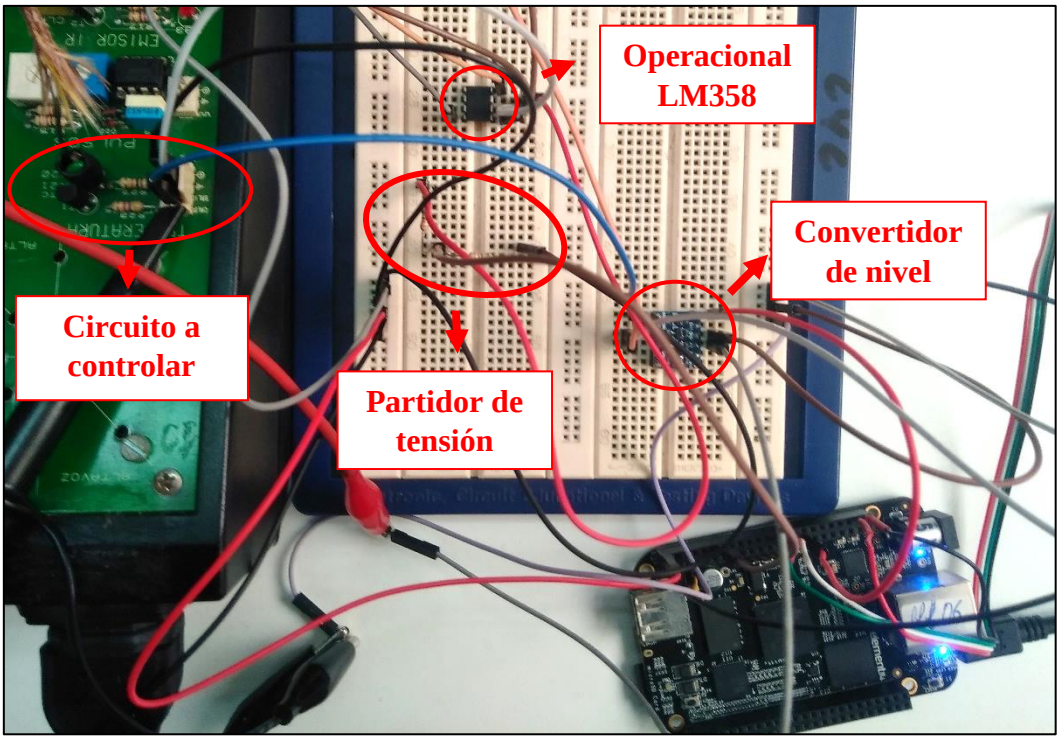

Ilustración 72. Circuito para realizar el control PID.

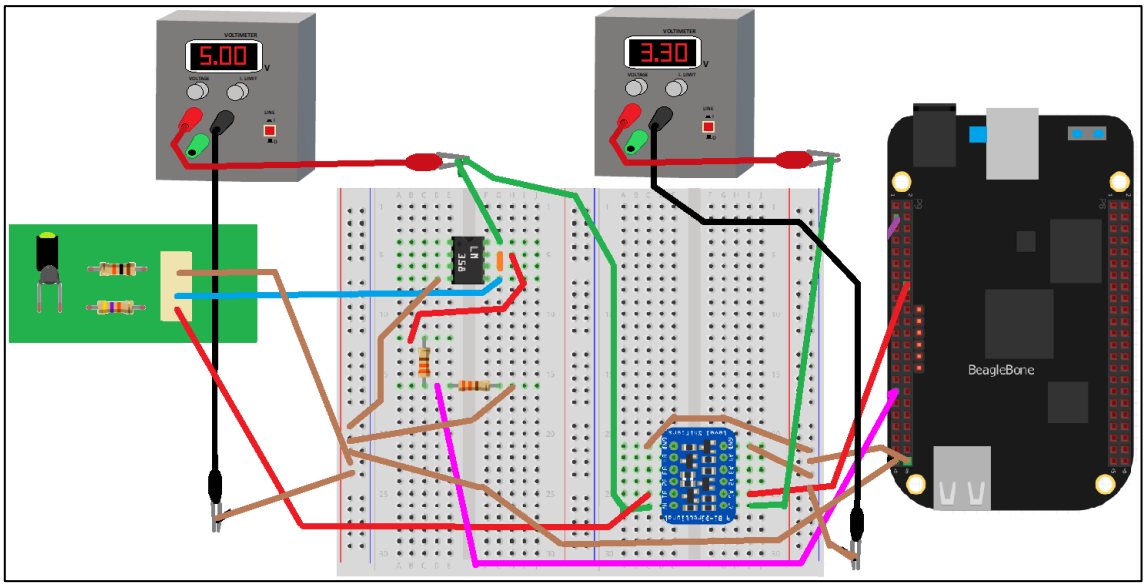

Ilustración 73. Esquema del circuito para realizar el control PID.

Una vez hemos montado el circuito y conseguido introducir un valor máximo de 1.8 V en la BeagleBone Black, el siguiente paso antes de meternos de lleno con el control PID, es buscar una relación entre el voltaje de la entrada analógica y el sensor de temperatura.

Lo que hemos hecho es ir alimentando el circuito con valores entre 0 y 3.3 V e ir midiendo que valores de temperatura se obtenían y que valores de tensión teníamos a la salida del partidor de tensión. De esta manera, hemos obtenido una tabla de valores para tratar de calibrar el sensor y ver la relación entre la temperatura y la tensión recibida. Se usó la herramienta GNU/Octave para representar los puntos, y buscar una posible función de ajuste. Vemos que es aproximadamente lineal. Haciendo un ajuste por mínimos cuadrados la ecuación final es:

$$T = m * V + b \tag{3.2}$$

Dónde el valor de m es 39.229 y el valor de b es -21.508.

Tensión (V)	Temperatura (ºC)
1.207	26
1.209	26
1.218	27
1.346	31
1.517	37
1.646	42
1.721	46
1.763	48
1.796	50

Tabla 9.Valores medidos para establecer una relación entre tensión y temperatura

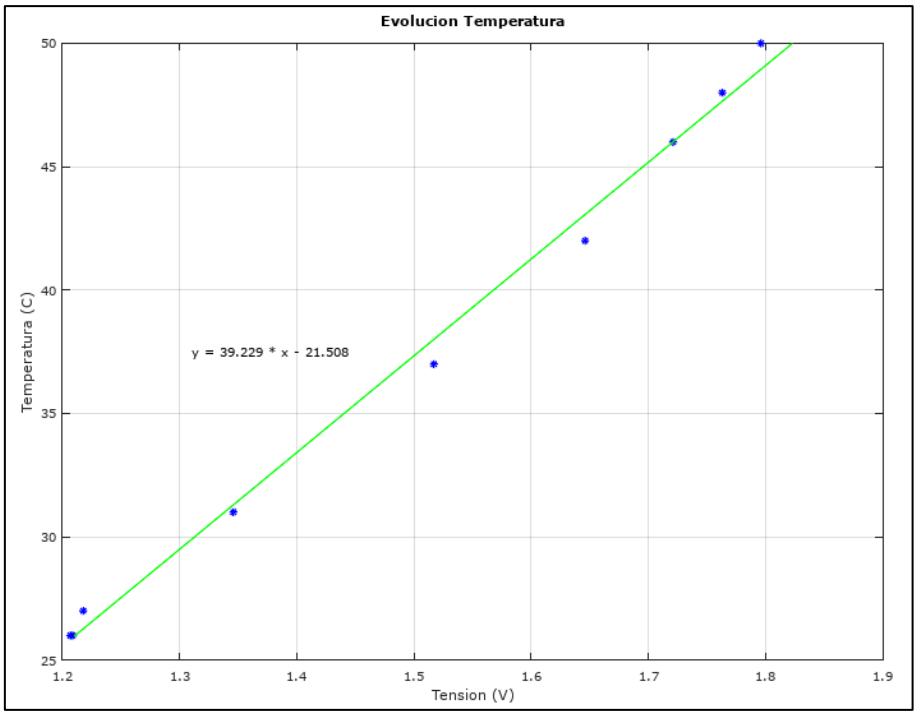

Ilustración 74. Recta que nos da la relación entre la temperatura (y) y la tensión (x).

Cuando ya hemos llevado a cabo los pasos previos, podemos pasar a realizar el control PID.

Hemos desarrollado una nueva clase que nos regule el control y los cálculos del PID y hemos elaborado 5 pruebas diferentes para realizar la comunicación entre PC y BBB consiguiendo así establecer dos configuraciones de sistemas de control distribuido mediante el bus CAN para el control de la temperatura mediante un regulador PID.

A continuación, se explicará la clase PID y se hará una descripción de las diferentes pruebas, las cuales corresponden con dos tipos diferentes de sistemas de control distribuido. Ambas pruebas se han desarrollado con varios hilos de ejecución concurrente para poder estar realizando dos tareas al mismo tiempo.

3.10.1. Clase PID

La clase PID tiene como objetivo realizar el control de temperatura de acuerdo con unos parámetros establecidos. Lo que se busca con esta clase es obtener el valor de salida del PID para unos valores preestablecidos y calculados.

Para poder desarrollar un control óptimo y deseado, los atributos que componen esta clase son:

- *double _Ts;* → guarda el tiempo de muestreo que se utilizará en el cálculo de la salida del PID.
- *int _consigna* → guarda el valor de la consigna.
- *double _Kp;* → establece el valor de la ganancia proporcional a aplicar sobre el controlador PID.
- *double _Ki;* → establece el valor de la ganancia integral a utilizar en el control PID.
- *double _Kd;* → mantiene el valor de la ganancia derivativa que vamos a aplicar sobre el regulador PID.
- *double _max;* → marca el máximo de PWM al que podemos llegar, en este caso será de 100 puesto que corresponde con el porcentaje de duty cycle.
- *double _min;* → establece el mínimo de valor que puede tener la PWM, al trabajar con porcentaje el mínimo será 0.

- *double _prerror;* → marca el error anterior al error actual, se utiliza para el cálculo del error en la parte derivativa y se va actualizando en cada ciclo.
- *double _integral;* → es el atributo utilizado para el cálculo del error integral.

Para completar el control PID, esta clase se compone de los siguientes métodos:

- *Pid (double Kp, double Ki, double Kd, double Ts, double max, double min);* → es el constructor de la clase, se encarga de recibir los valores de ganancia, muestreo, máximo y mínimo y guardarlos en sus atributos correspondientes para trabajar con ellos en el resto de los métodos.
- *calcular (double consigna, double salida);* → tiene el objetivo de calcular la parte proporcional, integral y derivativa, así como, cada uno de sus errores, para finalmente, con estos datos, poder calcular la PWM a aplicar para realizar el control del PID.
- *setConsigna (int consigna);* → guarda el valor de la consigna a utilizar en el PID.
- *setKp (double kp);* → obtiene la ganancia proporcional a utilizar en el PID.
- *setKi (double ki);* → guarda la ganancia integral que vamos a emplear en el control PID.
- *setKd (double kd);* → obtiene el valor de la ganancia derivativa para poder llevar a cabo el control PID del sistema.
- ~Pid ( ); → destructor de la clase.

Una vez hemos realizado la clase PID el siguiente paso es crear unas pruebas en las que se hará uso de ella para comprobar su correcto funcionamiento.

De esta manera, se plantearán dos tipos de sistemas distribuido dónde ambos están formados por la BBB y el PC, pero, se diferencian en la forma de trabajo.

3.10.2. Sistema de control distribuido 1

El primer sistema de control distribuido está pensado para que el PID se ejecute completamente en la BBB, mientras que el PC únicamente sirve para monitorizar el funcionamiento y cambiar parámetros. Así, la BeagleBone es la encargada de leer el valor analógico por sus entradas ADC, transformar este valor en temperatura, calcular la temperatura y la salida del circuito para pasar todos los datos al PC. El ordenador tendrá dos programas, en el primero se encargará de mostrar por pantalla el valor de estos datos, mientras que, en el segundo, pedirá al usuario si desea cambiar alguna de las ganancias, en caso de cambio, envía el nuevo valor de ganancia a la BBB, esta recibe los nuevos valores y aplica el PID. Esto se repite constantemente hasta que el usuario decida parar.

Un esquema gráfico de lo anteriormente explicado sería:

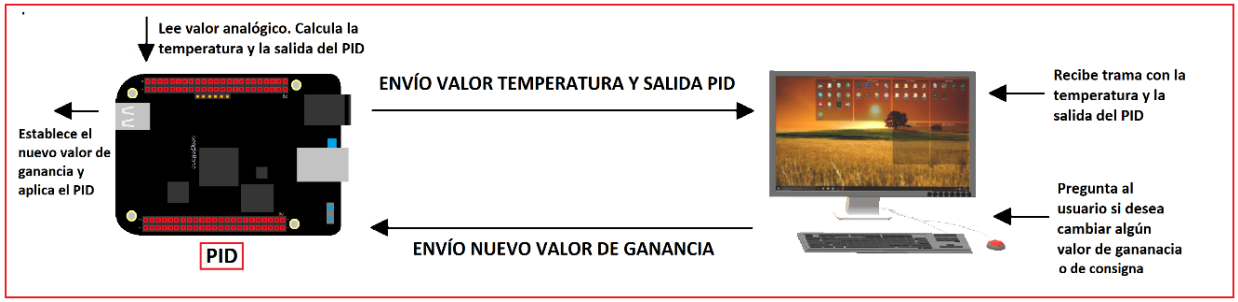

Ilustración 75. Sistema de control distribuido 1.

3.10.2.1 Prueba Envío

Para poder llevar a cabo esta prueba (ver *pruebaBBB_PID.cpp*) hemos creado un programa dónde trabajamos con hilos, es decir, de esta manera permitimos a la BeagleBone hacer dos procesos simultáneos:

- En un primer hilo, leemos los valores analógicos de la BBB, una vez hecho esto, calculamos la temperatura a la que se encuentra el circuito y la salida del control PID que es aplicada a la entrada del sistema a través de la PWM. Además, creamos dos tramas para enviarlas al PC para que reciba y muestre estos valores.
- El segundo hilo se encarga de recibir desde el PC unas tramas que contendrán, si el usuario así lo ha decidido, unos nuevos valores de ganancia. Cuando se obtienen estos valores, este hilo se encargará de aplicarlos para realizar el control PID.

Ambos hilos se ejecutarán infinitamente con los tiempos de acción establecidos para ir realizando el control PID de forma continua, hasta que el usuario decida pararlo.

3.10.2.2 Pruebas de Recepción

En cuanto a la prueba de recepción (ver *pruebaRecPC.cpp* y *pruebaRecGanPC.cpp*) estas se ejecutarán en el PC y sus objetivos consisten en:

- La primera prueba se encargará de ir recibiendo tramas con valores de temperatura y de la salida del PID enviadas desde la BeagleBone Black. Los identificadores que describen a estas tramas son los siguientes:
  - Identificador de temperatura:
  - Identificador de la salida:
- La segunda prueba pedirá por pantalla al usuario si desea cambiar alguno de los valores de las ganancias del sistema, y enviará una trama CAN con este nuevo valor a la BeagleBone Black para que lo aplique.

En este caso, las pruebas también se ejecutan infinitamente hasta que el usuario decida pararlo. En las siguientes imágenes podemos observar su funcionamiento:

```
[debian@bbb06 TFG_Itziar_CAN]$ sudo build/pruebaBBB_PID
        0        30   30.6038        0
        0        30   31.7863        0

El valor de consigna es: 35
   21.2944        35   31.9579   3.04206
   22.5724        35   32.6445   5.39753
   29.6864        35   32.3012   8.09629
   33.7489        35    32.492   10.6043
   35.9614        35   32.8925   12.7119
   38.8415        35   33.0832   14.6287
   35.8665        35   34.0558   15.5729
   37.8883        35   34.0368   16.5361
   37.1447        35   34.4182   17.1179
   36.7063        35   34.6471   17.4708
    33.674        35   35.1811   17.2897
   31.1758        35   35.4862   16.8035
   30.2034        35   35.4862   16.3173

El valor de Ki es: 1
   12.8277        35   35.5816   15.7357
   10.7585        35   35.8295   14.9061
   8.89912        35   36.0012    13.905
   9.04226        35   35.8105   13.0945

El valor de Ki es: 0
  0.238902        35   34.9522   13.1423
         0        35   35.6579   13.1423
         0        35   35.3909   13.1423
```

Ilustración 76. PID en la BBB.

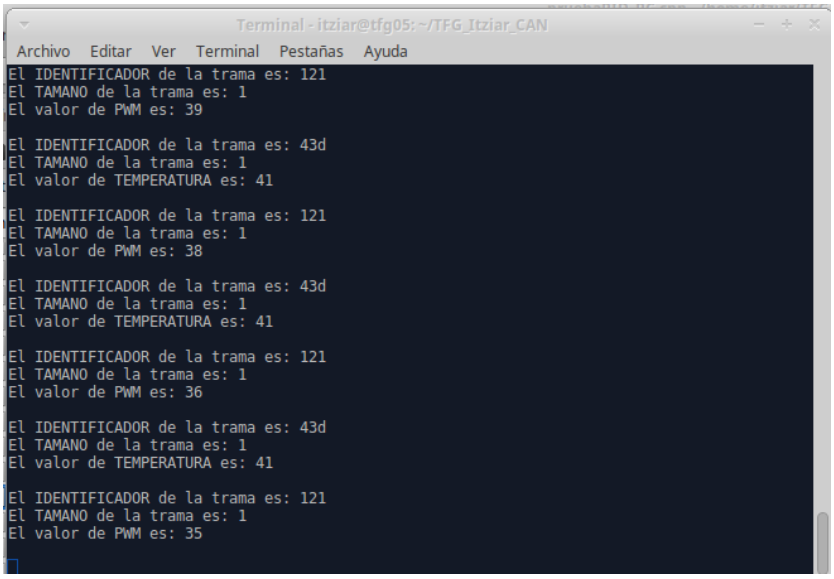

Ilustración 77. Recepción temperatura y pwm en PC.

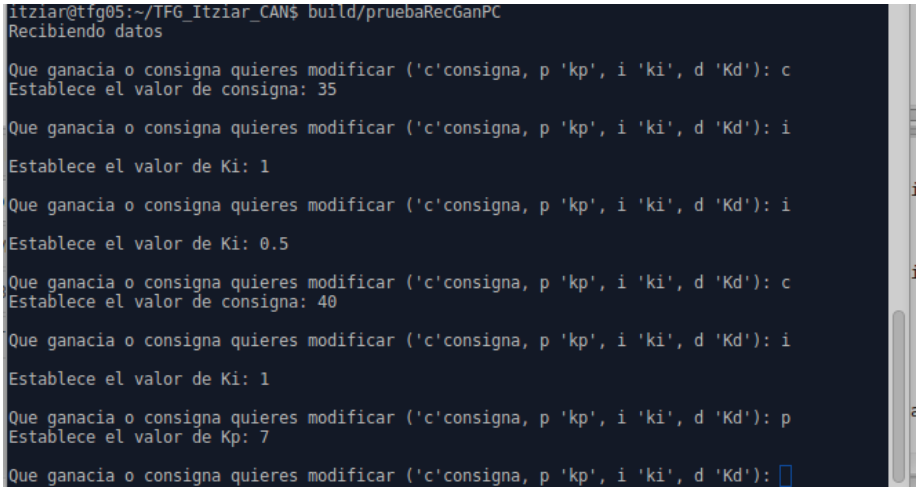

Ilustración 78. Modificar valor de consigna o de ganancias.

3.10.3. Sistema de control distribuido 2

El segundo sistema de control distribuido está pensado para que la BBB lea el valor analógico por sus entradas ADC y le envíe este valor al PC. El ordenador tendrá dos programas, en el primero se encargará de calcular la temperatura y salida del PID a partir del valor analógico recibido y aplicar el control PID, mientras que, en el segundo, pedirá al usuario si desea cambiar alguna de las ganancias. Esto se repite constantemente hasta que el usuario decida parar.

Un esquema gráfico de lo anteriormente explicado sería:

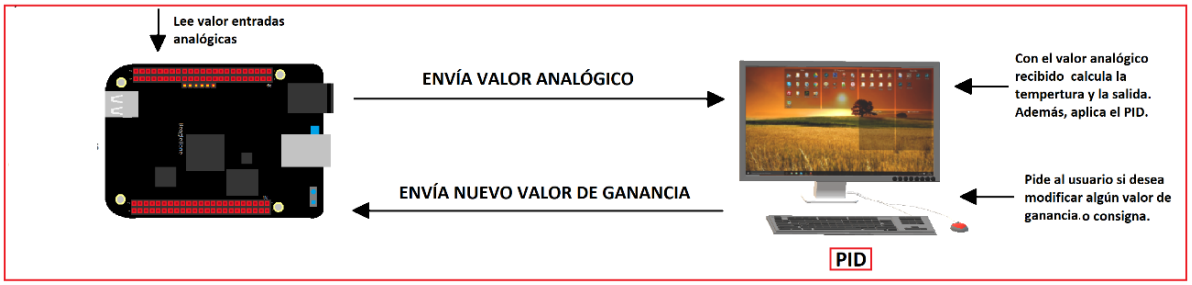

Ilustración X. Sistema de control distribuido 2.

3.10.3.1 Prueba Envío

Para poder llevar a cabo esta prueba (ver *pruebaEnvRecBBB.cpp*) también ha sido necesario llevar a cabo un programa con hilos, es decir, permitiendo a la BeagleBone hacer dos procesos simultáneos:

- En un primer hilo, leemos los valores analógicos de la BBB y los enviamos al PC a través de una trama CAN.
- El segundo hilo se encarga de recibir desde el PC unas tramas que contendrán, si el usuario así lo ha decidido, unos nuevos valores de ganancia. Cuando se obtienen estos valores, este hilo se encargará de aplicarlos.

Ambos hilos se ejecutarán infinitamente con los tiempos de acción establecidos para ir realizando el control PID de forma continua, hasta que el usuario decida pararlo.

3.10.3.2 Prueba Recepción

En cuanto a la prueba de recepción (ver *pruebaPID_PC.cpp* y *pruebaRecGanPC.cpp*) estas se ejecutarán en el PC y sus objetivos consisten en:

- La primera prueba se encargará de ir recibiendo las tramas con el valor analógico (con identificador: 5A0) de la entrada para utilizarlo en el cálculo de la temperatura y salida del PID y, además, llevará a cabo el control PID.
- La segunda prueba pedirá por pantalla al usuario si desea cambiar alguno de los valores de las ganancias del sistema, y enviará una trama CAN con este nuevo valor a la BeagleBone Black para que lo aplique.

En este caso, las pruebas también se ejecutan infinitamente hasta que el usuario decida pararlo.

En las siguientes imágenes podemos observar el funcionamiento del sistema de control distribuido descrito:

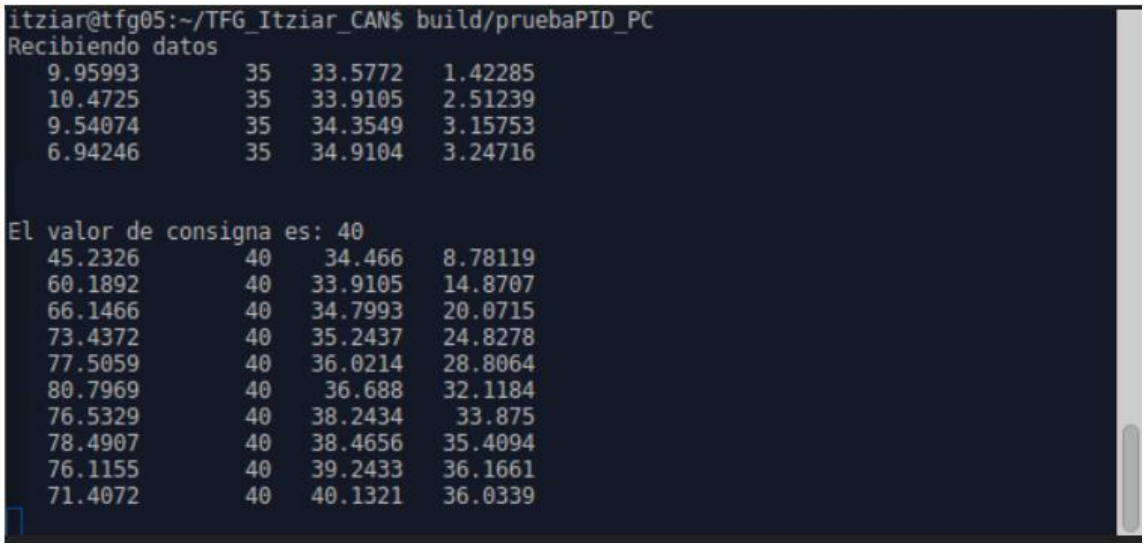

Ilustración 79. PID en el PC.

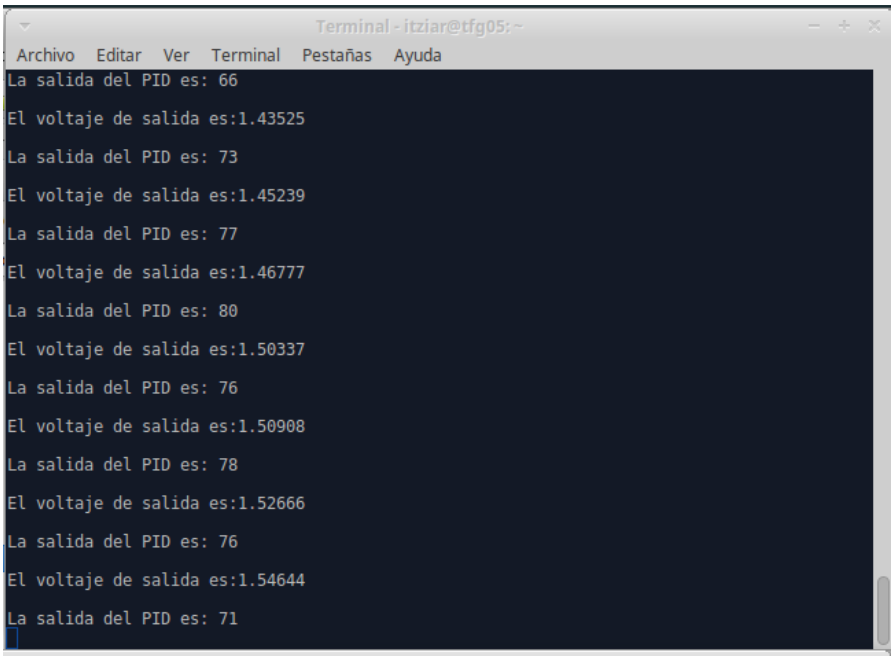

Ilustración 80. Recepción de datos en la BBB.

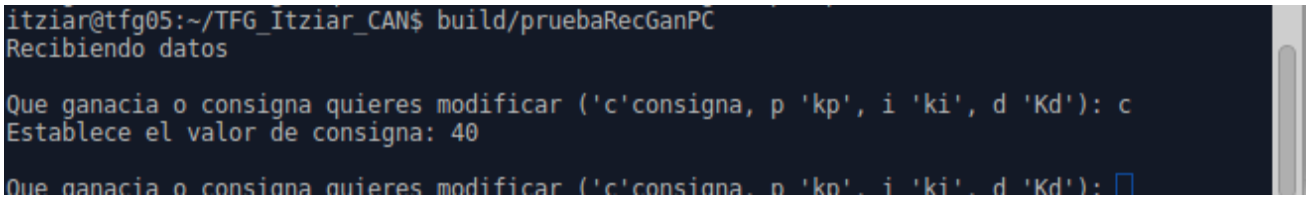

Ilustración 81. Pedir datos desde el PC.

3.11. Identificadores de las tramas

ID	TIPO DE MENSAJE	CODIFICACIÓN
555	Mensajes de datos aleatorios	Array de números entero
3A2	Mensajes de datos aleatorios	Array de números entero
1A3	Mensajes de datos aleatorios	Array de números entero
5A0	Valores analógicos de voltaje	Punto fijo con 3 decimales
12D	Porcentaje del duty cycle de la PWM	Número entero
123	Valor de período de la PWM	Número entero
43D	Valor de temperatura del circuito	Punto fijo con 3 decimales
121	Valor de salida del PID	Punto fijo con 3 decimales
54E	Valor de consigna	Número Entero
2B0	Ganancia Kp PID	Número entero
104	Ganancia Ki PID	Número entero
223	Ganancia Kd PID	Número entero

Tabla 10. Identificadores de las tramas.

Capítulo 4. Conclusions and Future Lines

4.1. Conclusions

In the development of the project it has been carried out a set of classes that have allowed us to control CAN's communication, ADC module, the BeagleBone Black's PWM, and the PID. It has been found that the creation of these classes make it easy the realization of programs for the user that allow him to use all these procedures. All clases have been developed with the idea that to manage the errors are launched exceptions.

We have implemented two distributed systems control for the PID's control that have been able to be developed thank to the BeagleBone, that through its module PWM, has allowed us to heat the resistance of the circuit we wanted to control, and the ADC module that has provided us information about the circuit temperature. In addition, it has been carried out the transmission and reception of data with another device connected to CAN bus. In our case, through a CAN card installed in the PC.

During the development of the project, the great power and performance presented by the BeagleBone Black has been verified. The use of this card has allowed us to develop a high number of processes quite different from each other, demonstrating its great versatility of work.

We have carried out programs with several threads of concurrent execution that have allowed us in the same test to be doing two different task.

The joint use of the BeagleBone Black and the CAN bus, as well as the elaboration of two distributed control systems has revealed the great couple that form both elements to be used in any control process, either industrial or non-industrial.

The advantages of speed, arbitrariness, distance, etc., provided by the CAN bus taking in consideration other communication buses, as well as the work capacity of the BBB, have shown us the importance of both elements separately and the great work they can do when they are combined.

4.2. Future Lines

Even though the project was finalized, since distributed control systems cover numerous processes, it can develop even more.

One option could be, in addition to controlling the temperature of a circuit, the control of speed of rotation of some blades as shown in the image.

Ilustración X. Control de giro.

Another option that could be made is to develop a control system with another BeagleBone Black and to carry out the distributed control between the two BeagleBone and the PC. This option would allow us to carry out works more similar to those of a big industry.

In addition, we could consider the idea of using another protocol of communication instead of the CAN protocol, an example could be the Ethernet protocol and to carry out a comparison between both to see the advantages and disadvantages of each one and which gives us better benefits.

Capítulo 5. Presupuesto

A continuación, se presentará el presupuesto del proyecto teniendo en cuenta unos costes indirectos en los materiales del 3% considerados ante posibles roturas o desgaste.

Materiales			
Elemento	**Número de elementos**	**Precio Unitario**	**Precio Total**
Ordenador con Ubuntu	1	320 €	320,00 €
BeagleBone Black	1	70,89 €	70,89 €
Tarjeta CPC_CPI	1	30 €	30,00 €
Cableado y conectores CAN	2	10 €	20,00 €
Protoboard	1	4,95 €	4,95 €
Potenciometro 1 KΩ	1	0,34 €	0,34 €
Led	1	0,48 €	0,48 €
Amplificador	1	0,25 €	0,25 €
Resistencias	3	0,21 €	0,63 €
Convertidor de niveles	1	2,52 €	2,52 €
Cables de conexión	15	0,05 €	0,75 €
Osciloscopio	1	342 €	342,00 €
Sonda	1	22 €	22,00 €
DISEN-EXP	1	50 €	50,00 €
Coste total materiales			**864,81 €**

Tabla 11. Coste de material.

Indirecto					
Elemento	**Número de elementos**	**Precio Unitario**	**Precio Total**	**Porcentaje CI**	**Coste Indirecto**
Ordenador con Ubuntu	1	320,00 €	320,00 €	3%	9,60 €
BeagleBone Black	1	70,89 €	70,89 €	3%	2,13 €
Tarjeta CPC_CPI	1	30,00 €	30,00 €	3%	0,90 €
Cableado y conectores CAN	2	10,00 €	20,00 €	3%	0,60 €
Protoboard	1	4,95 €	4,95 €	3%	0,15 €
Potenciometro 1 KΩ	1	0,34 €	0,34 €	3%	0,01 €
Led	1	0,48 €	0,48 €	3%	0,01 €
Resistencias	3	0,21 €	0,63 €	3%	0,02 €
Convertidor de niveles	1	2,52 €	2,52 €	3%	0,08 €
Amplificador	1	0,25 €	0,25 €	3%	0,01 €
Cables de conexión	3	0,05 €	0,15 €	3%	0,00 €
Osciloscopio	1	342,00 €	342,00 €	3%	10,26 €
Sonda	1	22,00 €	22,00 €	3%	0,66 €
DISEN-EXP	1	50,00 €	50,00 €	3%	1,50 €
Coste indirecto total					**24,43 €**

Tabla 12. Costes indirectos material.

Mano de obra			
Trabajo	**Número de horas**	**Precio por hora**	**Precio total**
Invesitgación y Programación	75	35 €	2.625 €
Desarrollo e Implantación	15	35 €	525 €
Coste total mano de obra			**3.150 €**

Tabla 13. Coste mano de obra.

Total proyecto	
Coste materiales	864,81 €
Coste mano de obra	3.150 €
Costes indirectos	24,43 €
Coste total Proyecto	**4.039,24 €**

Tabla 14. Coste total del proyecto.

Se ha de tener en cuenta que algunos precios como el de la tarjeta CPC_PCI o el de la mesa de control son estimados de productos con características similares debido a la dificultad de localización de su precio en el mercado.

Bibliografía

[1] Wikipedia, Sistemas de control distribuido. Disponible en la URL; https://es.wikipedia.org/wiki/Sistema_de_control_distribuido

[2] Wikipedia, Can Bus. Disponible en la URL: https://en.wikipedia.org/wiki/CAN_bus

[3] BeagleBone Black, página oficial. Disponible en la URL: https://beagleboard.org/black

[4] Wikipedia, BeagleBone Black. Disponible en la URL: https://es.wikipedia.org/wiki/BeagleBoard

[5] Dummies, BeagleBone vs BeagleBone Black. Disponible en la URL: https://www.dummies.com/computers/beaglebone/the-original-beaglebone-versus-the-beaglebone-black/

[6] Kernel SocketCAN. Disponible en la URL: https://www.kernel.org/doc/Documentation/networking/can.txt

[7] SocketCAN. Disponible en la URL: http://www.chuidiang.org/clinux/sockets/sockets_simp.php

[8] Can4linux - CAN network device driver. Disponible en la URL: http://www.can-wiki.info/can4linux/man/index.html

[9] SourceForge, Can4Linux CAN bus device driver. Disponible en la URL: https://sourceforge.net/projects/can4linux/

[10] Repositorio Git Hub para CAN4Linux. Disponible en la URL: https://github.com/linux-can/

[11] Repositorio Git Hub. ULL – InformaticaIndustrial - Empotrados TFG _ Itziar _ Rodríguez. Disponible en la URL: https://github.com/ULL-InformaticaIndustrial-Empotrados/TFG_Itziar_CAN

[12] Soporte de ayuda C++. Disponible en la URL: https://en.cppreference.com/w/

[13] Wikipedia, Control PID. Disponible en la URL: https://es.wikipedia.org/wiki/Controlador_PID

[14] Picuino, Control PID. Disponible en la URL: http://www.picuino.com/es/arduprog/control-pid.html

[15] Ilustración 1. Disponible en la URL: https://es.slideshare.net/estebankagelmacher/multiplexado

[16] Ilustración 2. Disponible en el: *Campus virtual ULL. Informática Industrial. Tema 5. Alberto F. Hamilton Castro.*

[17] Ilustración 3. Disponible en la URL: https://www.mindmeister.com/es/285805566/plantas-antigua-y-nuevas-plantas

[18] Ilustración 4 y 7. Disponible en la URL: https://www.intersil.com/content/dam/Intersil/whitepapers/rad-hard/using-can-bus-in-space-flight-applications.pdf

[19] Ilustración 5. Disponible en la URL: http://server-die.alc.upv.es/asignaturas/PAEEES/2005-06/A05-A06%20-%20Controlador%20CAN%20-%20Trabajo.pdf

[20] Ilustración 6. Disponible en la URL:
https://www.iso.org/obp/ui/#iso:std:iso:11898:-1:en

[21] Ilustración 8. Disponible en la URL:
Campus virtual ULL. Informática Industrial. Tema 6. Alberto F. Hamilton Castro.

[22] Ilustración 9. Disponible en la URL:
https://es.wikipedia.org/wiki/Bus_CAN

[23] Ilustración 10. Disponible en la URL:
https://electronilab.co/tienda/beaglebone-black-arm-cortex-a8-1ghz/

[24] Ilustración 11 y 12. Disponible en la URL:
https://ebookcentral-proquest-com.accedys2.bbtk.ull.es/lib/bull-ebooks/detail.action?docID=1882235

[25] Ilustración 13. Disponible en la URL:
https://www.innovadomotics.com/mn-tuto/mn-mod/mn-bgb/8-beagle.html

[26] Ilustración 15. Disponible en la URL:
https://es.wikipedia.org/wiki/Controlador_PID

[27] Ilustración 23. Disponible en la URL: https://es.aliexpress.com/item/5pcs-lot-4-channel-IIC-I2C-Logic-Level-Converter-Bi-Directional-Module-5V-to-3-3V/32794304980.html

[28] Olsson, Mikael., C++ 14 quick syntax reference, Apress, New York, Second Edition, 2015.

[29] Chacon, Scott y Straub, Ben., Pro Git, Apress, New York, First Edition, 2009. Consultado a través de: https://git-scm.com/book/es-ni/v1