ESCUELA SUPERIOR DE INGENIERÍA Y
TECNOLOGÍA
Grado en Ingeniería Electrónica Industrial y
Automática

Trabajo Fin de Grado

Realización de un sistema distribuido de control industrial utilizando Bus CAN

Autor: Jesús Alberto Hernández Bravo
Tutor: Alberto F. Hamilton Castro
Mayo, 2023

Resumen

El objetivo principal de este proyecto es desarrollar un sistema de comunicaciones basado en el bus CAN para ejercer el control distribuido de sistemas. Se utilizará el bus CAN para transmitir y recibir paquetes de datos entre una BeagleBone Black y un PC, y usando el sistema operativo Linux. Se desarrolarán programas y clases haciendo uso del lenguaje de programación C++ para enviar y recibir mensajes en ambas plataformas, aprovechando las capacidades de la librería “SocketCan”. Además, se explorará el uso de filtros para mejorar la comunicación y optimizar la selección de datos. Se estudiará la entrada analógica y salida PWM de la BeagleBone Black para interactuar con dispositivos externos. Por último, se llevará a cabo la creación de una se una serie de clases que implementen algoritmos de control, como control ON-OFF y control PID. Para demostrar el correcto funcionamiento del sistema, se aplicó a la planta *Process Controll Trainer 37-100*, que implementa el control de temperatura de aire.

Para el desarrollo de software se siguieron los paradigmas de la programación orientada a objetos. Se han desarrollado varias clases en C++ y una pequeña jerarquía en el caso de los controladores.

Abstract

The main objective of this project is to develop a communication system based on the CAN bus for device control within a network. The CAN bus will be utilized to transmit and receive data packets between a BeagleBone Black and a Linux-based PC. C++ programs will be implemented to send and receive messages on both platforms, taking advantage of the capabilities of the CAN bus. Additionally, the use of filters will be explored to improve communication and optimize data selection. The analog and PWM capabilities of the BeagleBone Black will be considered for integration, allowing interaction with external devices. Lastly, temperature control will be implemented using control algorithms such as ON-OFF and PID control, establishing communication between the BeagleBone and the PC via the CAN bus to transmit system control data. In summary, the project will focus on utilizing the CAN bus for effective communication, developing C++ programs for device control and implementing control algorithms over the network.

Índice

Indice de figuras

Abreviaturas y siglas

PLC	Controlador lógico programable
CAN	Red de área de controlador
OSI	Modelo de interconexión de sistemas abiertos
ISO	Organización Internacional para la Estandarización
CRC	Código de detección de errores
SOF	Campo de inicio de trama
ID	Campo de identificador de trama
DLC	Campo de longitud de datos
EOF	Campo de finalización de trama
POO	Programación orientada a objetos
BBB	BeagleBone Black
eMMC	Memoria flash integrada
RAM	Memoria de acceso aleatorio
UART	Transmisor-Receptor Asíncrono Universal
SPI	Interfaz periférica serial
I2C	Circuito inter-intergrado
ADC	Convertidor analógico digital
IoT	Internet de las cosas
GPIO	Entrada/Salida de Propósito General
PWM	Modulación por ancho de pulso
SSH	Secure Shell
PCI	Interconexión de componentes periféricos
API	Interfaz de programación de aplicaciones

PID	Control proporcional – integral - derivativo
Kp	Ganancia proporcional
Ki	Ganancia integral
Kd	Ganancia derivativa

Capítulo 1. Introducción

1.1. Objetivos

El objetivo central del proyecto consiste en profundizar en el conocimiento acerca del bus CAN y aplicarlo en la realización de un sistema de comunicaciones para llevar a cabo el control de dispositivos en una red. Para ello, haciendo uso del lenguaje de programación *C++*, se llevará a cabo el desarrollo de una serie de programas que hagan uso del protocolo CAN en los distintos dispositivos a controlar.

1.2. Redes de comunicaciones industriales

Las redes de comunicación son el pilar de cualquier sistema industrial, puesto que proporcionan un poderoso medio de intercambio de datos, así como proporcionar flexibilidad para conectar varios dispositivos. Actualmente tenemos diversos protocolos que nos permiten establecer un diálogo entre los distintos componentes de un sistema, dando como resultado la automatización y optimización de los procesos de control y, por lo tanto, una mejora en el funcionamiento de las industrias. Antes de la llegada de los sistemas de comunicaciones actuales, todo el control de los dispositivos se llevaba a cabo mediante una gran cantidad de cables que permitían transmitir las señales a grandes armarios, en los cuales mediante sensores y actuadores se lograba llevar a cabo la tarea de controlarlos, lo cual recibe el nombre de sistema de control centralizado.

Figura 1.1. Sistema de control centralizado en los años 60 [1]

Estos suponían una serie de inconvenientes tales como la gran cantidad de cables a usar, tanto para alimentar los dispositivos como para llevar a cabo la comunicación entre ellos. Pero todo cambió en el año 1968, el cual fue el año que dio el pistoletazo de salida a las redes de comunicación industrial, todo ello motivado por la salida al mercado del primer controlador lógico programable (PLC), siendo estos dispositivos capaces de detectar diversos tipos de señales del proceso, para luego elaborar y enviar acciones de acuerdo con lo que se ha programado. Las nuevas necesidades marcadas por estos dispositivos dieron lugar a la aparición de los primeros protocolos de comunicación y favorecieron que a principios de los años 1970 ya existieran los buses de campo [1].

1.2.1 Buses de campo

El punto de partida de la llegada de los buses de campo viene a raíz de la llegada de los PLC, y más tarde con la llegada del microprocesador, puesto que esto fomentó la creación de sistemas de comunicación industrial bidireccional y multipunto entre dispositivos de campo inteligentes, conociéndose como buses de campo. Es decir, un bus de campo es una red de área local dedicada a la automatización industrial, y enlaza los dispositivos de entrada (sensores, interruptores, etc.) y los dispositivos de salida (válvulas, accionamientos, lámparas de indicación, etc.) sin necesidad de conectar cada dispositivo individualmente al controlador (PLC, PC industrial, etc.). Antes de su introducción, los ordenadores se conectaban mediante conexiones seriales directas, por lo que sólo dos dispositivos podían comunicarse por conexión. El bus de campo, por otro lado, permite que cientos de puntos analógicos y digitales se conecten simultáneamente, ya sean del mismo fabricante o no. Esto reduce tanto el número de cables necesarios como la longitud de los mismos. [2].

1.2.2 Modelo OSI

Como consecuencia de la derivación de estos medios de comunicación en grandes y enormes sistemas complejos, en el año 1977, surge el modelo de interconexión de sistemas abiertos, también llamado OSI (en inglés *open system interconnection*), propuesto por la Organización Internacional para la Estandarización (ISO), siendo aprobado en el año 1984.

Es una normativa formada por siete capas que define las diferentes fases por las que deben pasar los datos para viajar de un dispositivo a otro sobre una red de comunicaciones. El modelo OSI establece una arquitectura jerárquica estructurada

donde se descompone el proceso complejo de la comunicación en varios problemas más sencillos, los cuales son asignados a las distintas capas, de forma que una capa no tenga que preocuparse por lo que hacen las demás. Según la estructura jerárquica, cada capa realiza servicios para la capa inmediatamente superior, a la que devuelve los resultados obtenidos, y a su vez demanda servicios a la capa inmediatamente inferior [3].

Figura 1.2. Capas del modelo OSI [4]

El modelo lo podemos subdividir en las siguientes capas:

• Capa **física**: Es la capa más baja del modelo y se encarga de la transmisión de bits a través de un medio físico, como un cable o una fibra óptica.

• Capa de **enlace de datos**: Esta capa se encarga de la transmisión de datos de manera fiable entre dispositivos conectados directamente, y de detectar y corregir errores en la transmisión.

• Capa de **red**: Esta capa se encarga de enrutar los paquetes de datos a través de la red utilizando protocolos como el IP, y de establecer y mantener la comunicación entre dispositivos.

• Capa de **transporte**: Esta capa se encarga de asegurar la transmisión de datos de manera fiable entre dos dispositivos finales, y de controlar el flujo y la congestión de la red.

• Capa de **sesión**: Esta capa establece, mantiene y finaliza las conexiones entre dispositivos, y coordina la comunicación entre aplicaciones.

• Capa de **presentación**: Esta capa se encarga de la representación y el formato de los datos para su intercambio entre dispositivos, y de la traducción de diferentes formatos de datos.

• Capa de **aplicación**: Esta capa se encarga de la interacción entre el usuario y las aplicaciones que utilizan la red, y proporciona servicios de red a las aplicaciones, como, por ejemplo; correo electrónico y navegación web.

1.3. BUS CAN

El BUS CAN fue desarrollado en la década de 1980 por la empresa alemana Bosch para satisfacer las necesidades de la industria automotriz en cuanto a comunicaciones en red. Antes del desarrollo del BUS CAN, los sistemas de comunicación en la industria automotriz eran muy limitados y no podían satisfacer las demandas de los nuevos sistemas electrónicos. El objetivo principal del BUS CAN era proporcionar una solución de comunicaciones confiable y robusta para los sistemas de control y automatización de los vehículos, que pudiera manejar grandes cantidades de datos y asegurar una comunicación segura y eficiente entre los diferentes sistemas electrónicos. Con el tiempo, el BUS CAN ha evolucionado y se ha extendido más allá de la industria automotriz, y hoy en día es utilizado en muchos otros campos, como la automatización industrial, la medicina, la aviación, la robótica y la electrónica en general.

El BUS CAN no se ajusta completamente al modelo OSI, ya que fue desarrollado antes de la creación del modelo y no todas sus capas están claramente definidas. Sin embargo, se puede relacionar la funcionalidad del bus con las capas del modelo OSI de la siguiente manera:

- La capa física de CAN se encarga de la transmisión de datos a través del medio físico, que generalmente es un cable bus. Esta capa se relaciona con la capa física del modelo OSI.
- La capa de enlace de datos de CAN se encarga de proporcionar un enlace de comunicaciones confiable, define el formato de los mensajes, el método de acceso al bus y se encarga de detectar errores en la transmisión de datos. Esta capa se relaciona con la capa de enlace de datos del modelo OSI.

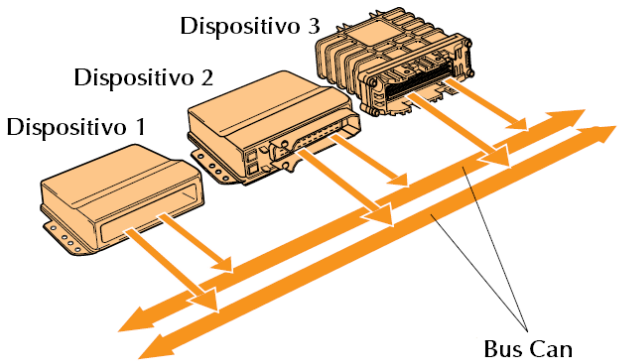

Figura 1.3. Representación BUS CAN [5]

En la figura 1.3, vemos que el protocolo CAN se basa en una topología bus, en la que todos los dispositivos de la red están conectados a un mismo canal de comunicación, también conocido como "bus". En esta topología, la información se transmite en ambos sentidos a través del bus y todos los dispositivos tienen acceso a ella. Los mensajes se envían a través del bus en modo difusión, es decir, para todos los nodos, aunque solo los nodos interesados pueden leer el mensaje enviado. Además, es importante aclarar que la comunicación de los mensajes se produce de manera simultánea a todos los dispositivos, garantizando así la sincronización, a diferencia de otras topologías que usan conexiones punto a punto, donde el mensaje no llega a todos los dispositivos a la vez. Otra parte importante es que los mensajes tienen un identificador, el cual indica el tipo de información que contiene, y no el destinario del mismo como ocurre en otras redes. La topología de bus es una de las topologías más simples y económicas de implementar, pero también tiene la desventaja de que, si se produce una falla en el bus, toda la red puede verse afectada. Este tipo de arquitectura permite la comunicación en tiempo real entre dispositivos, lo que es esencial en aplicaciones de control y automatización. Una de las características más importantes de CAN es su capacidad para detectar errores en la transmisión de datos. Cada mensaje transmitido por un dispositivo incluye un código de detección de errores (CRC), que se utiliza para comprobar la integridad de los datos recibidos. Si se detecta un error en la transmisión, el dispositivo receptor puede solicitar la retransmisión del mensaje. Otra característica importante de CAN es su alta velocidad de transmisión. El protocolo original, conocido como CAN 2.0, puede transmitir datos a velocidades de hasta 1 Mbps. Sin embargo, se han desarrollado versiones posteriores del protocolo, como CAN FD (*Flexible Data-Rate*), que pueden alcanzar velocidades de transmisión de hasta 8 Mbps.

1.3.1 Componentes en el BUS CAN

En primer lugar, los distintos dispositivos conectados a la red se conocen con el nombre de nodos, siendo estos los que quieren transmitir y recibir datos para comunicarse entre sí. Los nodos pueden ser sensores, actuadores, unidades de control… etc. En segundo lugar, estos nodos están interconectados por medio de un cableado de par trenzado blindado para la transmisión de señales. El cableado debe ser de alta calidad para garantizar la integridad de las señales y evitar interferencias electromagnéticas. En tercer lugar, cada nodo tiene incorporado un controlador, que es el dispositivo que controla la transmisión y recepción de datos a través del BUS CAN, el cual puede ser un microcontrolador, un microprocesador o un circuito integrado dedicado. En cuarto lugar, el dispositivo que convierte la señal de datos del controlador en una señal adecuada para la transmisión a través del BUS CAN se conoce con el nombre de transceptor. También se encarga de la recepción de señales del BUS CAN y las convierte en señales digitales que el controlador pueda entender. Por último, se utilizan resistencias colocadas al final

del bus, conocidas como terminadores, con el objetivo de reducir la reflexión de señales y garantizar una transmisión de datos más confiable.

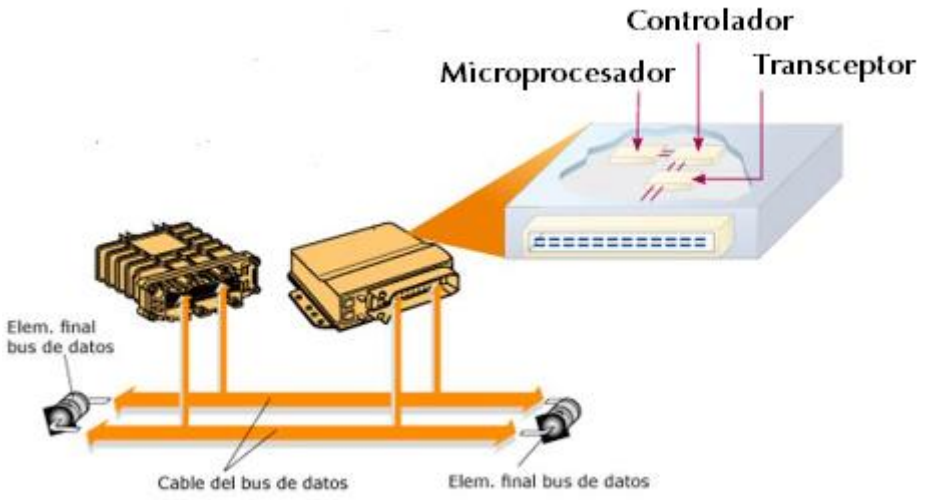

Figura 1.4 Componentes BUS CAN [5]

1.3.2 Capa física CAN

La capa física del BUS CAN es la capa más baja del protocolo CAN y se encarga de la transmisión de los bits de datos a través de este. Esta utiliza un esquema de comunicación diferencial, lo que significa que la señal transmitida es la diferencia de voltaje entre dos cables en lugar de una señal absoluta en un solo cable. Esto es importante porque ayuda a minimizar el ruido y la interferencia electromagnética que puede afectar la calidad de la señal. Además, la capa física del BUS CAN define las características eléctricas y mecánicas del bus, incluyendo el tipo de cableado, la impedancia de los cables, la longitud máxima del bus y la velocidad de transmisión de datos. *CAN High* y *CAN Low* son las dos señales de cableado que se utilizan para la comunicación diferencial. La diferencia de voltaje entre las dos señales representa el valor de datos que se está transmitiendo. En la transmisión de datos, el transmisor envía la señal diferencial a través de la línea CAN High y la línea CAN Low. El receptor recibe la señal y determina el valor de los bits de datos por la diferencia de voltaje entre las dos señales, como podemos observar en la figura 1.4.

En el estándar CAN, se especifica que el rango de voltaje diferencial mínimo para representar un "0" lógico es de 2 voltios (V). Si la diferencia de voltaje es menor a 2V, se considera que se está transmitiendo un "1" lógico. En cuanto a la velocidad de transmisión de datos, el bus CAN puede operar a velocidades de hasta 1 Mbps (megabits por segundo) en redes de corta distancia y hasta 125 kbps (kilobits por segundo) en redes de larga distancia.

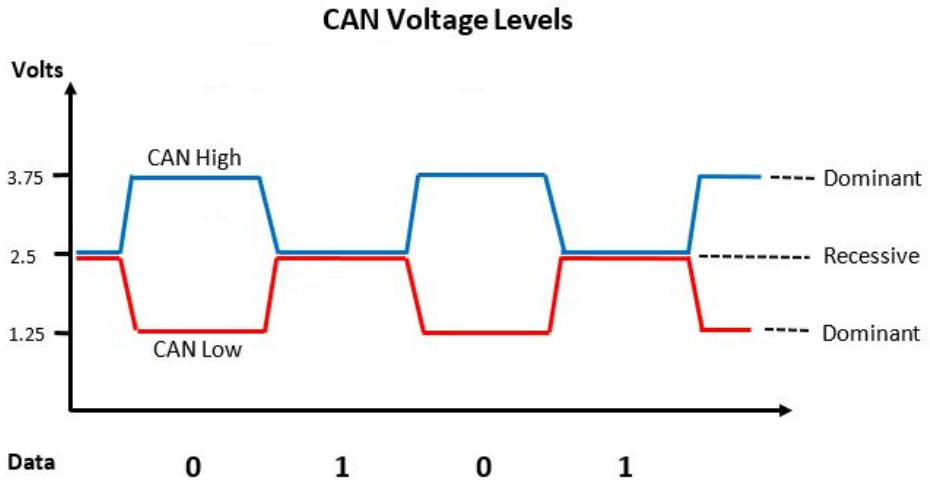

Figura 1.5 CAN High y CAN Low [7]

Además, el "0" se considera dominante mientras que el "1" se considera recesivo. Esto significa que si algún dispositivo envía un "0" y otro dispositivo envía un "1", se hace la y-lógica y prevalecerá el "0" dominante.

1.3.3 Capa de enlace CAN

Por lo tanto, la manera en la que los dispositivos de la red se van a comunicar va a ser a través de paquetes de datos conocidos como tramas. Las tramas CAN están diseñadas para ser eficientes, confiables y flexibles, y están compuestas por distintos campos que incluyen:

• Campo de **inicio de trama** (SOF): es un bit dominante que indica el inicio de una trama.

• Campo de **identificador de trama** (ID): indica el identificador de mensaje y su prioridad.

• Campo de **longitud de datos** (DLC): indica el número de bytes de datos que se transmiten en la trama.

• Campo de **datos** (*Data Field*): contiene los datos que se transmiten.

• Campo de **CRC**: se utiliza para detectar errores en la transmisión de datos.

• Campo de **fin de trama** (EOF): indica el final de una trama.

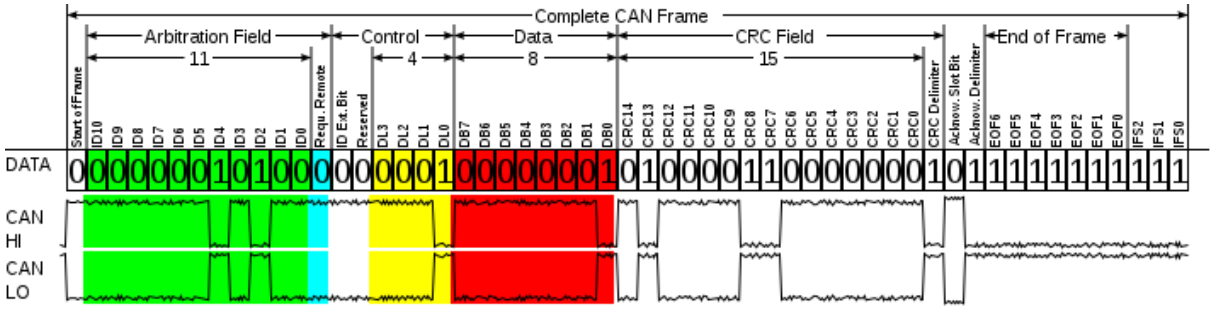

Figura 1.6 Trama CAN [6]

Además, las tramas CAN pueden ser de dos tipos: tramas de datos y tramas de control o remotas. Las tramas de datos se utilizan para transmitir información real entre los nodos del BUS CAN, y este tipo de trama es la representada en la figura 1.6, mientras que las tramas de control se utilizan para solicitar información o enviar comandos.

Las tramas CAN también utilizan un esquema de arbitraje, donde el mensaje que tiene el identificador más bajo, es decir, con mayor cantidad de "0", transmite su trama en el BUS CAN. La comunicación se realiza en serie y de manera asíncrona. La comunicación asincrónica implica que los dispositivos que se comunican no están sincronizados en términos de tiempo. En este tipo de comunicación, los dispositivos se comunican de manera no simultánea, es decir, no necesitan estar operando al mismo tiempo.

Por lo tanto, el arbitraje es el proceso que se utiliza para resolver conflictos en el bus. Múltiples nodos pueden intentar enviar mensajes simultáneamente en el mismo momento. En tales situaciones, el BUS CAN utiliza el arbitraje para determinar cuál no tiene el derecho de transmitir su mensaje en el bus. El proceso de arbitraje en el BUS CAN es el siguiente:

En primer lugar, cuando un nodo desea transmitir un mensaje, primero verifica si el bus está libre. Si lo está, el nodo comienza a transmitir el mensaje. El mensaje comienza con una identificación de mensaje única, que indica el tipo de mensaje y su prioridad. Mientras el nodo transmite el ID, los demás nodos en el bus también escuchan el ID. Si dos o más nodos comienzan a transmitir mensajes simultáneamente, sus ID pueden colisionar en el bus, y cuando ocurre una colisión, se produce el proceso de arbitraje, y finalmente transmite su trama aquel cuyo ID sea más bajo. Este nodo tiene prioridad y puede transmitir su mensaje después de la colisión. Después de que el nodo complete su transmisión, los otros nodos pueden volver a intentar la transmisión de su mensaje. Este proceso se repite cada vez que ocurre una colisión, y es importante destacar que el proceso de arbitraje en el BUS CAN se lleva a cabo en tiempo real, lo que significa que los mensajes se transmiten y reciben rápidamente en la red. Además, el proceso de arbitraje se realiza sin la necesidad de un controlador centralizado, lo que hace que el BUS CAN sea altamente confiable y eficiente. También es importante destacar que se trata de un proceso no destructivo, es decir, siempre se envía una trama, a

diferencia de Ethernet, por ejemplo, donde se pierden los paquetes implicados que se transmiten simultáneamente.

1.4. BUS CAN en GNU/Linux

GNU/Linux es un sistema operativo de código abierto y gratuito, basado en el núcleo Linux y en las herramientas del proyecto GNU. GNU es un acrónimo recursivo que significa "*GNU No es Unix*". Fue elegido nombre del proyecto para expresar la idea de que el sistema operativo GNU no estaba basado en Unix, pero tenía una funcionalidad similar. Además, GNU hace referencia a la imagen del animal gnú (gnu en inglés), que es un animal de la familia de los antílopes y que se utiliza como mascota del proyecto GNU. Unix es un sistema operativo multiusuario y multitarea, diseñado originalmente en los años 70 en los laboratorios Bell de AT&T en Estados Unidos. Unix es conocido por su estabilidad, eficiencia y seguridad, y ha sido ampliamente utilizado en servidores, estaciones de trabajo y otros sistemas informáticos. Este ha tenido una gran influencia en el desarrollo de otros sistemas operativos modernos, incluyendo Linux, macOS y Android. Además, muchos de los conceptos y herramientas utilizadas en Unix, como la interfaz de línea de comandos y las tuberías, siguen siendo fundamentales en la informática actual.

GNU/Linux fue creado en el año 1991 por el programador finlandés Linus Torvalds como un sistema operativo para computadoras personales, pero rápidamente se convirtió en un sistema operativo popular para servidores, dispositivos embebidos y otros sistemas informáticos. La filosofía de GNU/Linux se basa en la idea de que el software debe ser gratuito y accesible para todos, y que la comunidad de desarrolladores debe trabajar juntos para crear y mantener el software de manera colaborativa y transparente. Como resultado, el sistema operativo GNU/Linux es altamente personalizable, escalable y flexible. GNU/Linux se compone de varias partes clave, incluyendo el núcleo Linux, las herramientas y bibliotecas del proyecto GNU, y un conjunto de aplicaciones y herramientas de terceros. El núcleo Linux (Kernel) es el componente central del sistema operativo que se encarga de la comunicación con el hardware y el control de los recursos del sistema. El proyecto GNU proporciona una amplia variedad de herramientas y bibliotecas esenciales para el sistema operativo, incluyendo el *shell* de línea de comandos, las bibliotecas estándar del sistema y muchas otras herramientas. Una de las principales ventajas de GNU/Linux es su capacidad para personalizar y modificar el sistema operativo según las necesidades de los usuarios y las organizaciones. Los usuarios pueden modificar el código fuente y crear sus propias versiones personalizadas del sistema operativo, lo que permite adaptarlo a sus necesidades específicas. Otra ventaja importante de GNU/Linux es su seguridad y

estabilidad. Debido a su arquitectura y diseño, GNU/Linux es menos propenso a los virus y a los problemas de seguridad que otros sistemas operativos. Además, la naturaleza de código abierto del sistema operativo significa que cualquier vulnerabilidad o problema de seguridad puede ser rápidamente detectado y corregido por la comunidad de desarrolladores.

1.4.1 SocketCAN

El protocolo CAN es compatible con los controladores de dispositivos de Linux. Principalmente existen dos tipos. Controladores basados en dispositivos de caracteres y controladores basados en *sockets* de red. Un socket de red es una interfaz que permite a los programas de software comunicarse a través de una red utilizando diferentes protocolos de comunicación [8]. El kernel de Linux admite CAN, con el paquete SocketCAN, que proporciona una interfaz de socket para las aplicaciones del espacio del usuario, que se basa en la capa de red de Linux.

SocketCAN es un conjunto de controladores de red y una API (Interfaz de programación de aplicaciones) que proporciona una interfaz para el bus CAN en sistemas operativos Linux. Con SocketCAN, los programadores pueden interactuar con dispositivos CAN conectados a una máquina Linux utilizando una API de sockets de red, lo que facilita el desarrollo de aplicaciones y herramientas de software que utilizan el bus CAN. SocketCAN es compatible con muchos tipos de controladores de red CAN, incluyendo aquellos basados en hardware y controladores basados en software, y permite a los programadores acceder a la información del bus CAN en tiempo real.

Una de las principales ventajas de SocketCAN es su capacidad para proporcionar una API de programación unificada para diferentes controladores CAN. Esto significa que los programadores pueden escribir aplicaciones de bus CAN independientemente del hardware utilizado, lo que facilita la portabilidad del software a diferentes sistemas. Además, SocketCAN también proporciona funciones de configuración y diagnóstico, lo que facilita la configuración de dispositivos CAN y la solución de problemas de la red. [12]

Es decir, básicamente podemos entender de manera mas sencilla que SocketCAN se encargará de la comunicación entre el programa, y el bus CAN.

Existen varios comandos que utilizan el socket CAN para enviar y recibir datos en un bus CAN en sistemas Linux. Algunos de ellos son:

1. cansend: este comando se utiliza para enviar un mensaje en el bus CAN. El mensaje se especifica en formato hexadecimal y se puede especificar la identificación del mensaje (ID) y la longitud de los datos.
2. candump: este comando se utiliza para capturar y mostrar todos los mensajes que se reciben en un bus CAN.

3. cangen: este comando se utiliza para generar mensajes aleatorios y enviarlos al bus CAN. Se pueden especificar la ID del mensaje y la longitud de los datos.
4. can-utils: esta es una colección de herramientas que incluye varios comandos para trabajar con el bus CAN, como candump, cansend...

Estos comandos y herramientas utilizan el socket CAN para acceder al bus CAN y enviar y recibir mensajes.

1.5. Dispositivos del proyecto

1.5.1 BeagleBone Black

BeagleBone Black (BBB) es una placa de desarrollo de bajo costo basada en el procesador ARM Cortex-A8 de 32 bits con una frecuencia de reloj de 1 GHz. Fue diseñada por BeagleBoard.org como una plataforma de prototipado para proyectos de electrónica y robótica. La placa tiene 512 MB de memoria RAM, 4 GB de almacenamiento flash, Ethernet, HDMI, USB host y dispositivo, y una variedad de puertos de entrada y salida que incluyen UART, SPI, I2C y ADC. Lo que hace que BeagleBone Black sea única es su capacidad de conectarse a una amplia variedad de dispositivos y periféricos externos, gracias a sus puertos GPIO y la compatibilidad con diferentes buses de comunicación como I2C y SPI. Esto hace que sea una placa muy versátil para proyectos de IoT y robótica. Además, la BeagleBone Black tiene una gran comunidad de desarrolladores y entusiastas detrás de ella, lo que significa que hay una gran cantidad de recursos y proyectos disponibles en línea para los usuarios que deseen experimentar con la placa.

BeagleBone Black es compatible con varios sistemas operativos, incluyendo Linux, de hecho, se envía con una distribución de Linux preinstalada en la tarjeta eMMC (memoria flash integrada), que incluye una variedad de herramientas y software para el desarrollo de proyectos. Además dispone de 96 pines de conexión, la mayoría con hasta 8 funcionalidades distintas.

Dentro de estos pines, tiene soporte para el bus CAN. Es necesario configurar la comunicación CAN en la BeagleBone Black a través del sistema operativo Linux que ejecuta la placa. La configuración incluye definir los pines de entrada/salida que se usarán para la comunicación CAN, la velocidad de transmisión, el formato de los mensajes CAN, entre otros. BeagleBone Black incluye los controladores CAN integrados en su Kernel de Linux, pero no incluye transceptores CAN en la placa base.

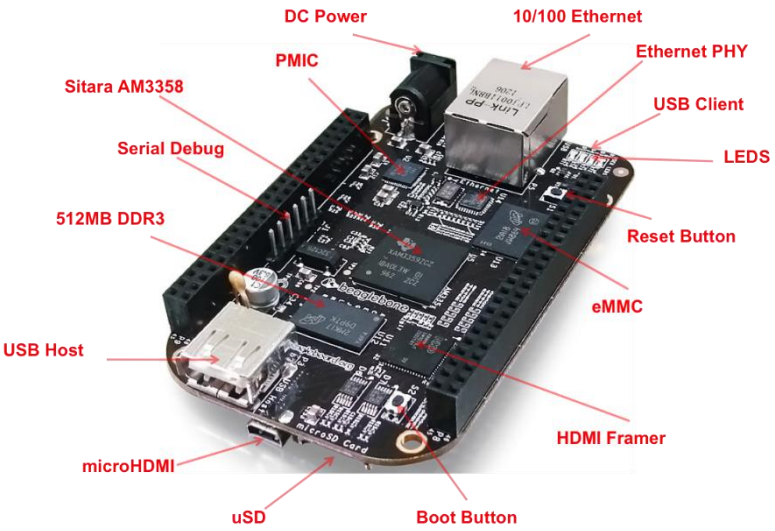

Figura 1.7 Componentes BeagleBone Black [9]

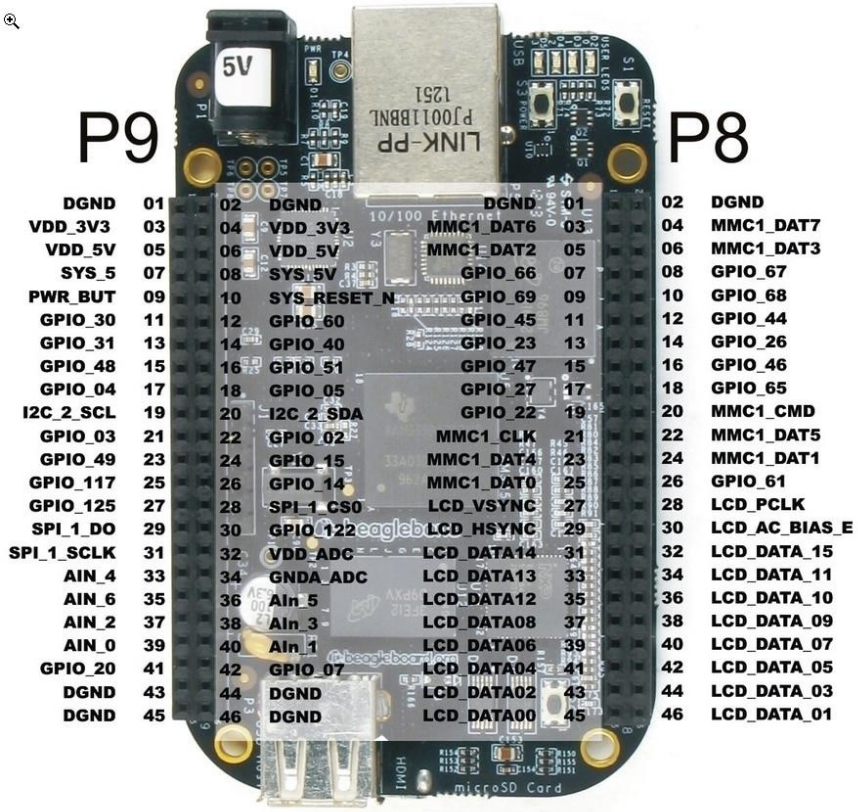

Figura 1.8 Pines BeagleBone Black [10]

La BeagleBone Black dipone de funcionalidades en sus pines tales como ADC, PWM y CAN, que se utilizaran en este proyecto. El ADC (conversor analógico digital), añade capacidad de entrada analógica a BeagleBone Black, permitiendo que la placa mida señales analógicas en lugar de solo señales digitales. El ADC tiene ocho canales de entrada analógica y un conversor analógico a digital de alta precisión que permite la medición de voltajes de entrada en un rango de 0 a 1,8V y con una resolución de 12 bits. Por otra parte, la funcionalidad de la PWM (modulación por ancho de pulso) añade capacidad de salida PWM a BeagleBone Black, permitiendo que la placa genere señales de PWM con una alta resolución y precisión. Esta funcionalidad tiene cuatro canales de salida PWM, cada uno de los cuales puede generar señales de PWM en una variedad de frecuencias y rangos de resolución. Además, BeagleBone Black tiene dos controladores CAN incorporados en su hardware: el controlador CAN0 y el controlador CAN1. Ambos controladores son compatibles con el protocolo CAN 2.0 y están disponibles en los pines de la placa. El controlador CAN0 está disponible en los pines P9_19 (CAN0_TX) y P9_20 (CAN0_RX), mientras que el controlador CAN1 está disponible en los pines P9_24 (CAN1_TX) y P9_26 (CAN1_RX). En el controlador CAN, el pin TX se utiliza para transmitir los datos y el pin RX se utiliza para recibirlos.

El comando “config-pin” es una herramienta específica de la distribución de Debian para la plataforma BeagleBone Este comando se utiliza para configurar los pines en la BeagleBone Black de una manera más conveniente. Este permite asignar diferentes modos y funcionalidades a los pines de la BeagleBone Black. Por ejemplo, puedes configurar un pin como entrada, salida, habilitar una función específica como I2C, SPI, UART, etc.

Si se ha iniciado sesión en la BBB y se escribe el comando config-pin sin ningún argumento, este muestra una lista de los pines disponibles en la BeagleBone Black y su configuración actual, incluyendo si están configurados como GPIO, UART, I2C, etc. Esto proporciona una visión general de los modos operativos actuales de los pines GPIO en la BBB.

1.5.2 Ordenador con CPC-PCI

El otro dispositivo que vamos a usar se trata de un ordenador que incorpora dos tarjetas CPC-PCI. Es una tarjeta de interfaz que se utiliza para conectar una PC a una red CAN. Esta tarjeta es compatible con el bus PCI (Interconexión de componentes periféricos) y proporciona dos puertos CAN con conectores DB9. Estos puertos pueden configurarse para trabajar en diferentes velocidades de transferencia de datos. La tarjeta CPC-PCI CAN es compatible con una amplia gama de sistemas operativos, siendo Linux el que será utilizado en este proyecto. Esto la hace adecuada para una amplia variedad de aplicaciones de automatización y control, desde pruebas de prototipos hasta aplicaciones industriales. [13]

1.6. C++

Puesto que el objetivo del proyecto se centra el uso del bus CAN, se van a crear una serie de clases y programas en C++ que permitan trabajar con el bus en diversos dispositivos, y por tanto, se hará una breve introducción acerca de este. C++ es un lenguaje de programación de alto nivel que se utiliza principalmente para desarrollar software de sistema, software de aplicaciones, software de servidores y software de dispositivos empotrados. Fue desarrollado por Bjarne Stroustrup en 1983 como una extensión del lenguaje C. La historia de C++ se remonta a los años 70, cuando el lenguaje C fue desarrollado por Dennis Ritchie. El lenguaje C se convirtió rápidamente en un lenguaje popular para el desarrollo de sistemas operativos y aplicaciones de bajo nivel debido a su eficiencia y capacidad de acceso directo a los recursos de hardware. Sin embargo, el lenguaje C carecía de ciertas características orientadas a objetos que se estaban desarrollando en ese momento. En 1983, Bjarne Stroustrup, un programador de *Bell Labs*, desarrolló C++ como una extensión del lenguaje C para agregar características orientadas a objetos. Stroustrup quería crear un lenguaje que pudiera combinar las características de bajo nivel de C con la facilidad de uso y la capacidad de abstracción de la programación orientada a objetos. El nombre "C++" se derivó de la notación "++" que se usa en programación para indicar el incremento de una variable en una unidad. C++ es un lenguaje compilado, lo que significa que el código fuente se traduce a un lenguaje de bajo nivel que la computadora puede entender. Esto permite que los programas escritos en C++ sean altamente eficientes y rápidos en comparación con otros lenguajes de programación interpretados. Una de las principales características de C++ es su soporte para programación orientada a objetos (POO), lo que permite a los desarrolladores escribir código que se puede reutilizar fácilmente y mantener de manera más sencilla. Además, C++ también admite programación genérica, que permite a los desarrolladores crear algoritmos y estructuras de datos que son independientes del tipo de datos. Otras características importantes de C++ incluyen su capacidad para manejar memoria dinámica, su soporte para sobrecarga de operadores y su amplio conjunto de bibliotecas estándar. C++ se ha utilizado en una amplia variedad de aplicaciones, desde sistemas operativos y controladores de dispositivos hasta aplicaciones de juegos y aplicaciones empresariales, estableciendo 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. A razón de las características mencionadas anteriormente se ha decidido utilizar este lenguaje de programación para llevar a cabo los distintos programas que se van a realizar en el proyecto.

Capítulo 2. Desarrollo del sistema de comunicación.

2.1. Pruebas de envío y recepción.

Sabiendo que este proyecto tiene como propósito desarrollar un sistema de comunicación basado en el BUS CAN que permita la interconexión de diferentes dispositivos, lo primero que haremos será utilizar la BeagleBone Black para realizar una prueba de envío y recepción de datos usando los dos controladores CAN incorporados en su hardware; el controlador CAN0 y el controlador CAN1.

Se comenzará por hacer una prueba de envío y recepción de datos, utilizando el CAN0 como transmisor y el CAN1 como receptor. Para ello utilizaremos los comandos "cangen" y "candump". Ambos comandos son parte de la suite de herramientas de SocketCAN, que como ya hemos visto es una implementación del protocolo CAN para sistemas Linux, y en la que profundizaremos más adelante. El comando "cangen" se utiliza para generar mensajes de prueba en el bus CAN. Con este comando, se pueden generar mensajes aleatorios o personalizados, además de poder especificar la velocidad de transmisión y el identificador del mensaje que se desea generar.

Por otra parte, el comando "candump" permite ver los mensajes que están siendo enviados y recibidos en el bus CAN. Con este comando se pueden ver todos los mensajes que se envían y reciben en tiempo real, lo que es muy útil para la depuración de problemas en el sistema. También se pueden filtrar los mensajes para ver solo aquellos que cumplan ciertos criterios específicos.

Lo primero que haremos es usar el comando "ssh", que es una herramienta de la línea de comandos que se utiliza para conectarse a un dispositivo remoto a través de una red utilizando el protocolo SSH (Secure Shell). SSH es un protocolo de red que proporciona una forma segura de acceder a un dispositivo remoto y ejecutar comandos de forma segura. Para realizar la conexión a la BeagleBone Black utilizando SSH se seguirán una serie de pasos. En primer lugar, hay que conectar la placa directamente a un ordenador donde queremos establecer la conexión remota a través de un cable USB. Luego, hay que encender la BeagleBone Black y esperar a que se inicie correctamente, para abrir una terminal o línea de

comandos en el ordenador y ejecutar el comando "ssh" seguido de la dirección IP de la BeagleBone Black y el nombre de usuario que se quiere utilizar para conectarse. Para ello se abren dos interfaces para cada controlador:

- *ssh debian@192.168.6.2*

por otra parte, en otro terminal se abre también una interfaz, pero esta vez usando la dirección 192.168.7.2. Por lo tanto, ya se han abierto dos interfaces para realizar una prueba de envío y recepción de datos.

A continuación, se tienen que configurar los pines correspondientes para usarlos como transmisores y receptores can, y para ello usamos los comandos:

- *config-pin P9_19 can*
- *config-pin P9_20 can*

siendo el pin 19 el que actuará como receptor, y el pin 20 el que actuará como transmisor del controlador CAN0. Este mismo proceso se repite en este caso para los pines 24 y 26, para ser configurados como receptor y transmisor del CAN1 respectivamente.

Por otra parte, hay que establecer la velocidad a la cual se transmitirán los datos, y para ellos se usa el comando:

- *sudo ip link set can0 up type can bitrate 125000*

llevando a cabo el mismo proceso para el CAN1. Se aclara lo que hace cada parte del comando:

El comando "sudo" se utiliza para ejecutar el siguiente comando con privilegios de administrador, "ip link set" es el comando principal que se utiliza para configurar las interfaces de red en Linux, "can0" especifica el nombre de la interfaz CAN que se está configurando, "up" indica que se está activando la interfaz CAN (si la interfaz no se activa, no se pueden enviar ni recibir mensajes CAN a través de ella), "type can" especifica que se está configurando una interfaz CAN, y por último "bitrate 125000", que especifica la velocidad de transmisión de la interfaz CAN en bits por segundo. En este caso, se está configurando la velocidad

de transmisión en 125,000 bits por segundo. Esta velocidad es común en redes CAN y es adecuada para muchas aplicaciones.

Por último, es necesario efectuar el levantamiento de los dos controladores usando el comando:

- *sudo ifconfig can0 up*

de nuevo usando el comando sudo, y también para el otro controlador CAN1. Ya estaría preparada la configuración para comenzar a realizar pruebas de envío de tramas desde una de las dos interfaces, pero antes se tiene que realizar la conexión física entre los pines de ambos controladores como se muestra en la figura 2.1

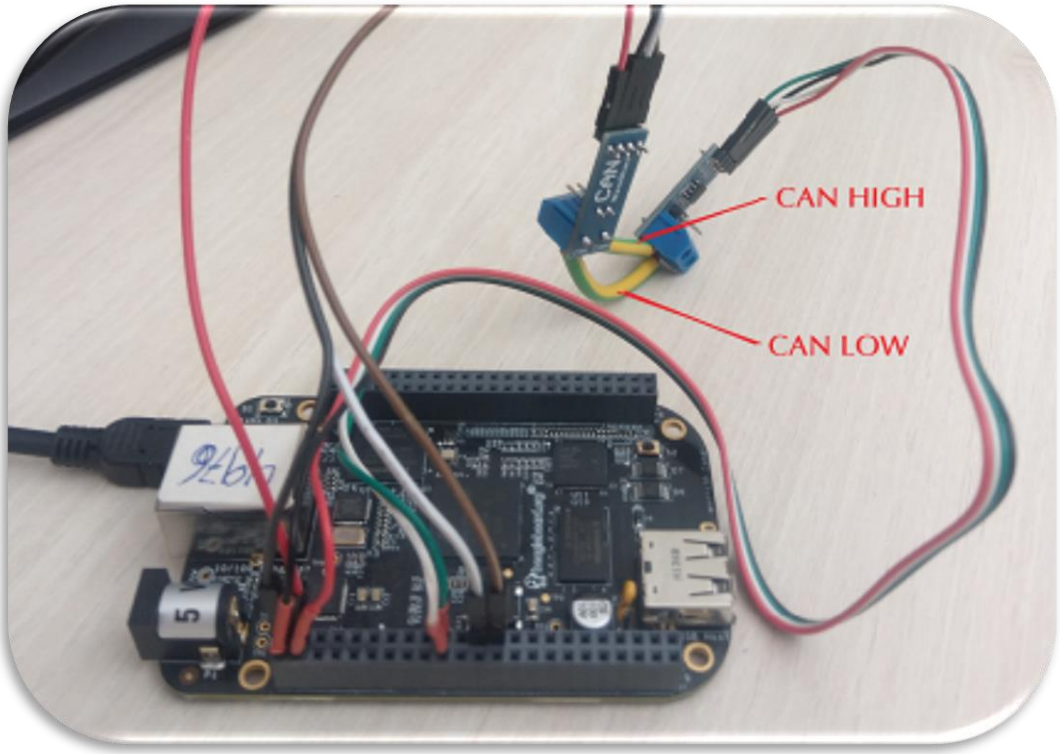

Figura 2.1. Conexión física BBB

Sabiendo que la BBB dispone de los controladores CAN, pero no de los transceptores necesarios, hay que conectar un transceptor a cada controlador, así como conectarlos a la alimentación de 3.3V que proporciona la Beagle Bone en los pines P9-3 y P9-4, uno para cada transceptor, así como a tierra, en los pines P9-1 y P9-2. Entre los dos transceptores conectamos dos cables, que serán el CAN High y el CAN Low del bus. Los transceptores para utilizar en el proyecto serán los SN65HVD230 de Texas Instruments, como podemos observar en la figura 2.2.

Figura 2.2 Transcepor SN65HVD230 [11]

A continuación, se utiliza uno de los dos interfaces abiertos para utilizar el comando:

- *candump can0*

configurando el controlador CAN1 en modo pasivo, a la espera de recibir tramas, y el comando:

- *cangen can1*

para enviar tramas aleatorias desde el controlador CAN0 a el controlador CAN1, como se puede observar en la figura 2.4

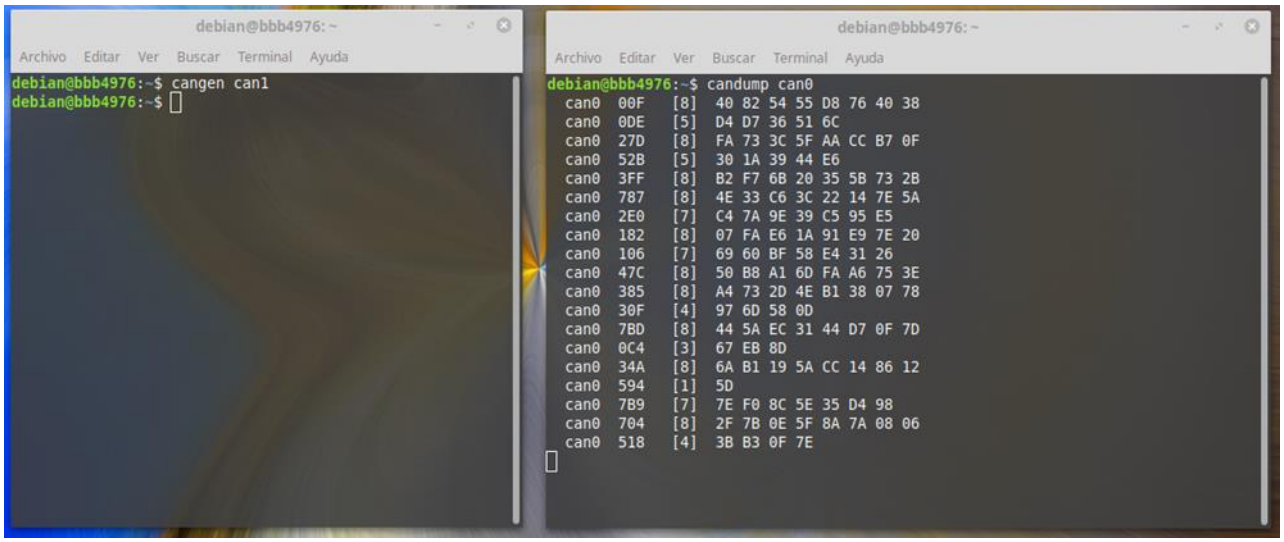

Figura 2.3. Prueba envío y recepción BBB.

Como podemos observar el comando "cangen" ha ido generando paquetes aleatorios de tramas de datos. Cada trama CAN enviada a través del comando "cangen" consta de las siguientes partes:

• Identificador de trama: el ID de trama es un número de identificación único, que como se aclara en el apartado 1.3.2 se utiliza para identificar una trama CAN. El ID de trama puede tener 11 bits (para identificación de trama estándar) o 29 bits (para identificación de trama extendida) de longitud. El ID de trama se especifica como un número hexadecimal de 3 o 8 dígitos (para identificación estándar o extendida, respectivamente) después del comando "cangen".

• Longitud de la trama (DLC): la longitud de la trama indica el número de bytes de datos que se están enviando en la trama CAN. La longitud de la trama se especifica como un número decimal de 1 a 8 después del ID de trama.

• Datos de la trama (Data): los datos de la trama son los bytes de datos que se están enviando a través de la red CAN. Los datos de la trama se especifican como un número hexadecimal de 1 a 8 bytes después de la longitud de la trama.

Por otra parte, vamos a realizar la misma prueba de envío y recepción, pero esta vez para el ordenador. Como se comenta con más profundidad en el apartado 1.5, este ordenador dispone de una tarjeta CPC-PCI que se utiliza para conectar una PC a una red CAN. En la figura 2.4, se muestra la tarjeta, así como dos conectores DB9, uno para cada puerto CAN.

Figura 2.4 Tarjeta CPC-PCI.

Para llevar a cabo la prueba en el ordenador, será necesaria la apertura de dos terminales, una para cada controlador CAN, que también se designan desde el sistema con el nombre de CAN0 y CAN1. Para llevar a cabo la preparación de ambos controladores, solo tenemos de que establecer el bitrate, y activar ambos puertos CAN, es decir:

- *sudo ip link set can0 up type can bitrate 125000*
- *sudo ifconfig can0 up*

Y los dos mismos comandos para el puerto CAN1. De esta manera, y habiendo conectado físicamente ambos puertos como se muestra en la figura 2.5, se puede comenzar a realizar una prueba de envío de y recepción de datos

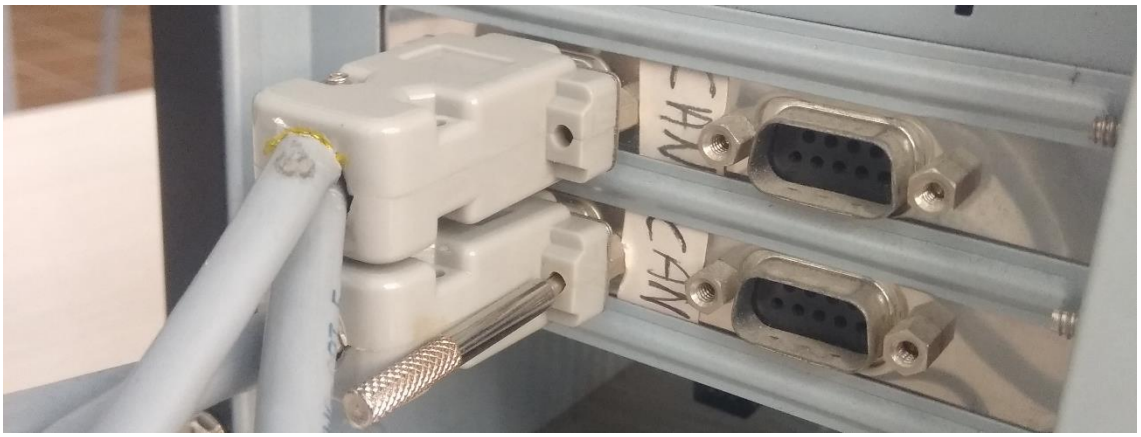

Figura 2.5 Conexión controladores en PC.

Por lo tanto, al igual que antes vamos a usar el comando “cangen” para generar paquetes aleatorios desde el controlador CAN0, y usamos el comando “candump” para recibir los paquetes dirigidos a CAN1.

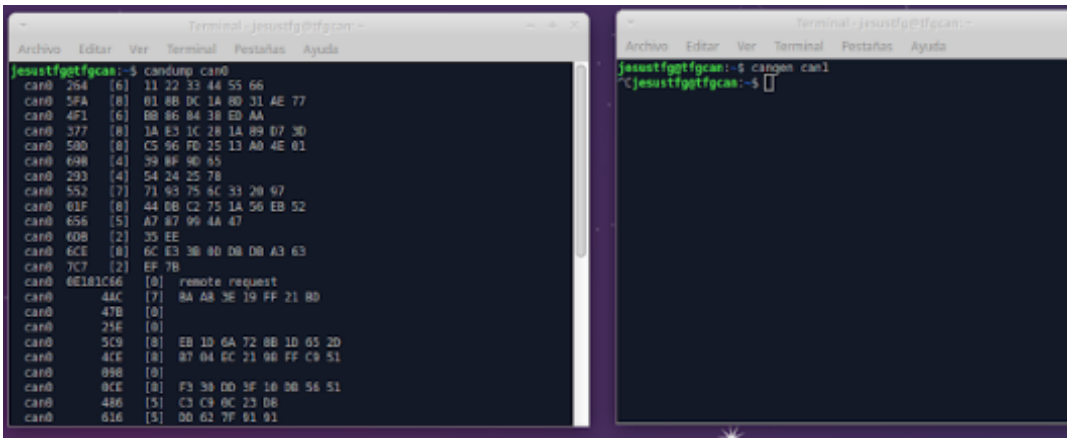

Figura 2.6 Prueba envío y recepción PC.

2.2. Pruebas de envío y recepción desarrolladas en C++.

En lugar de utilizar las herramientas de línea de comandos "candump" y "cansend" que proporciona SocketCAN, se ha utilizado la página web mencionada como referencia [12] para crear dos programas en C++ que realizan las mismas funciones. Estos programas permiten una mayor flexibilidad y control en la programación de la comunicación CAN en sistemas Linux. El programa de envío de paquetes CAN presenta una interfaz de programación clara para crear paquetes de datos personalizados y enviarlos a través de una interfaz de red CAN. Por otro

lado, el programa de recepción de paquetes proporciona una forma de escuchar y analizar los paquetes de datos que se reciben en la interfaz de red CAN. Al desarrollar estos programas personalizados, se puede adaptar la comunicación CAN a necesidades específicas de una aplicación, lo que puede mejorar la eficiencia y la fiabilidad del sistema en general, y será lo que se hará más adelante en el proyecto.

2.2.1 Programa de envío

Como se aclara en la página web [12], podemos encontrar todas las librerías necesarias para llevar a cabo el código de envío, así como las funciones de cada parte de este. El objetivo del programa es enviar un paquete CAN a través de una interfaz de red CAN, utilizando la API de SocketCAN. En la figura 2.7 se puede observar el código completo, pero antes se hará una descripción de cada función del mismo:

- La primera parte del programa se encarga de abrir un socket de comunicación en modo raw, lo que significa que los datos se transmiten en su forma original sin modificaciones, lo que puede ser importante para aplicaciones donde se necesita un control total sobre la estructura del paquete de datos. Luego, se declara la estructura de socketCAN y una estructura de red que se utiliza para conectar el socket a una interfaz de red CAN.

- A continuación, se configura la estructura “ifr” con el nombre de la interfaz can1, y se utiliza la función “bind” para asociar el socket creado con la dirección y puerto locales especificados en la estructura addr. Es decir, se une el socket a la dirección y puerto local.

- Después se configura el “loopback” para que el paquete no se reciba en la propia interfaz con “setsockopt”.

- Luego, se establece una estructura de paquete CAN con un identificador de mensaje en hexadecimal (0x649), un número de bytes de datos (2) y los datos específicos que se van a enviar (0x11 y 0x22).

- Finalmente, se llama a la función “write” para enviar el paquete CAN a través de la interfaz de red CAN vinculada al socket. Si la operación de envío es exitosa, se muestra un mensaje indicando que el paquete fue enviado correctamente. Si no es así, se muestra un mensaje de error.

Es decir, el programa primero abre un socket CAN en modo raw, luego lo vincula a la interfaz can1, y a continuación, configura el paquete que se desea enviar, que en este caso es un mensaje CAN con un identificador de 11 bits (0x649) y dos bytes de datos (0x11 y 0x22), y finalmente se envía el paquete a través del socket.

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <linux/can.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/can/raw.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <unistd.h>

int main(){

    // Declaración del socket
    int s;
    // Declaración de la estructura de un socket CAN
    struct sockaddr_can addr;
    // Declaración de estructura de red
    struct ifreq ifr;

    // Apertura de Socket modo raw
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

    // Comprobar que se abrió bien
    if ( s == -1 ){
        std::cout << "El socket no se ha abierto correctamente" << std::endl;
        return 2;
    } else {
        std::cout << "El socket se ha abierto correctamente" << std::endl;
    }

    // Unir socket a dirección y puerto local
    strcpy(ifr.ifr_name, "can1");
    ioctl(s, SIOCGIFINDEX, &ifr);

    addr.can_family = AF_CAN;
    addr.can_ifindex = ifr.ifr_ifindex;

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

    // Retroalimentación
    int loopback = 0;
    setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));

    // Paquete a enviar
    struct can_frame frame;
    frame.can_id = 0x649;
    frame.can_dlc = 2;
    frame.data[0] = 0x11;
    frame.data[1] = 0x22;

    // Envío de paquete
    int nbytes = write(s, &frame, sizeof(struct can_frame));
    if (nbytes == -1) {
        std::cout << "Error al enviar el paquete" << std::endl;
    } else {
        std::cout << "Paquete enviado correctamente (" << nbytes << " bytes)"
        << std::endl;
    }

    // Cerramos socket
    close(s);

    return 0;
```

Figura 2.7 Programa de envío desarrollado en C++

2.2.2 Programa de recepción

El objetivo del programa es abrir un socket CAN en modo raw y esperar a recibir tramas en el bus CAN. Cuando se recibe una trama, muestra la información relevante sobre la trama. El programa se divide en varias secciones:

• En la sección principal, se declara el socket y la estructura de dirección del socket CAN. A continuación, se crea el socket en modo raw y se comprueba si se ha abierto correctamente. Luego, se une el socket a la dirección y puerto local. Se establece la opción de retroalimentación en el socket, para que los paquetes enviados no se reenvíen a sí mismos.

• En la sección de recepción, se declara la estructura de trama CAN y se utiliza la función "read" para recibir tramas del bus CAN. Si la función "read" no recibe tramas, el programa se queda en un bucle esperando a recibir una. Si se recibe una trama, se muestra el número de bytes recibidos, el número de bytes del paquete, el identificador de la trama y los datos del paquete.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <linux/can.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/can/raw.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <unistd.h>

int main(){
    //Declaración del socket
    int s;
    //Declaración de la estructura de un socket CAN
    struct sockaddr_can addr;
    //Declaración de la estructura de red
    struct ifreq ifr;

    //Apertura de socket modo raw
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

    //Comprobar que se abrió bien
    if ( s == -1 ){
        std::cout << "El socket no se ha abierto correctamente" << std::endl;
        return 2;
    } else {
        std::cout << "El socket se ha abierto correctamente" << std::endl;
    }

    //Unir socket a dirección y puerto local
    strcpy(ifr.ifr_name, "can1" );
    ioctl(s, SIOCGIFINDEX, &ifr);
```

```
    addr.can_family = AF_CAN;
    addr.can_ifindex = ifr.ifr_ifindex;

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

    //Retroalimentación
    int loopback = 0;

    setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));

    //Recibir tramas
    struct can_frame frame;
    ssize_t nbytes = read(s, &frame, sizeof(struct can_frame));

    /* paranoid check... */
    while(1){
        nbytes = 0;

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

        if((nbytes < 0) || ((unsigned) 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;
            }
        }
    }

    return 0;
}
```

Figura 2.8 Programa de recepción desarrollado en C++

2.2.3 Pruebas de envío y recepción en PC

En la figura 2.9 se ilustra la ejecución del programa de envío, y en la figura 2.10. el de recepción.

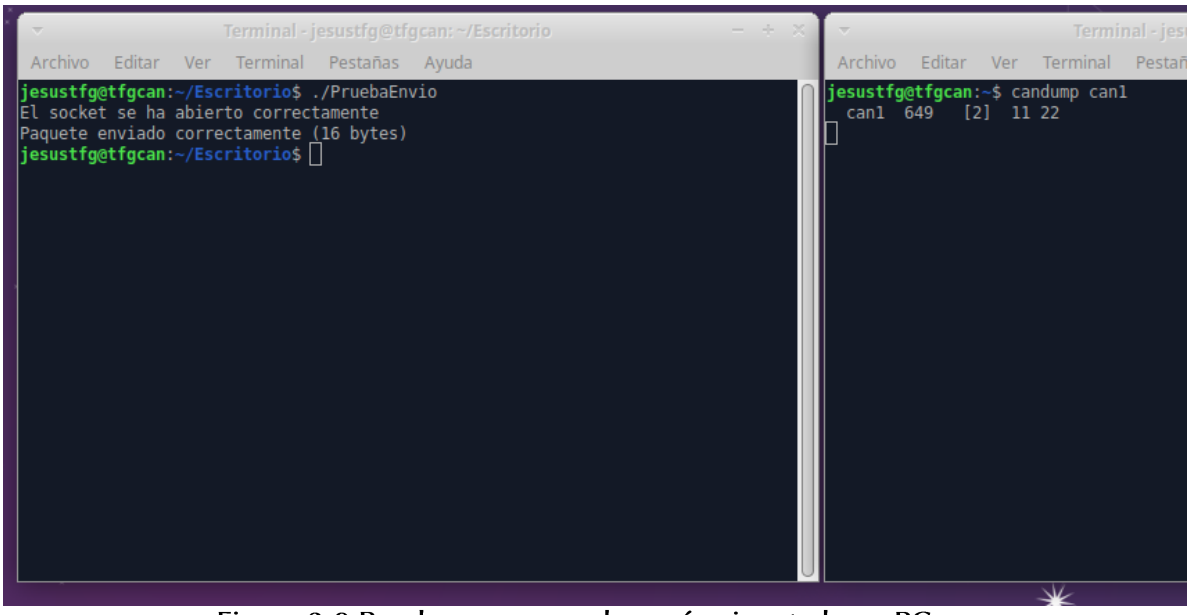

Figura 2.9 Prueba programa de envío ejecutada en PC

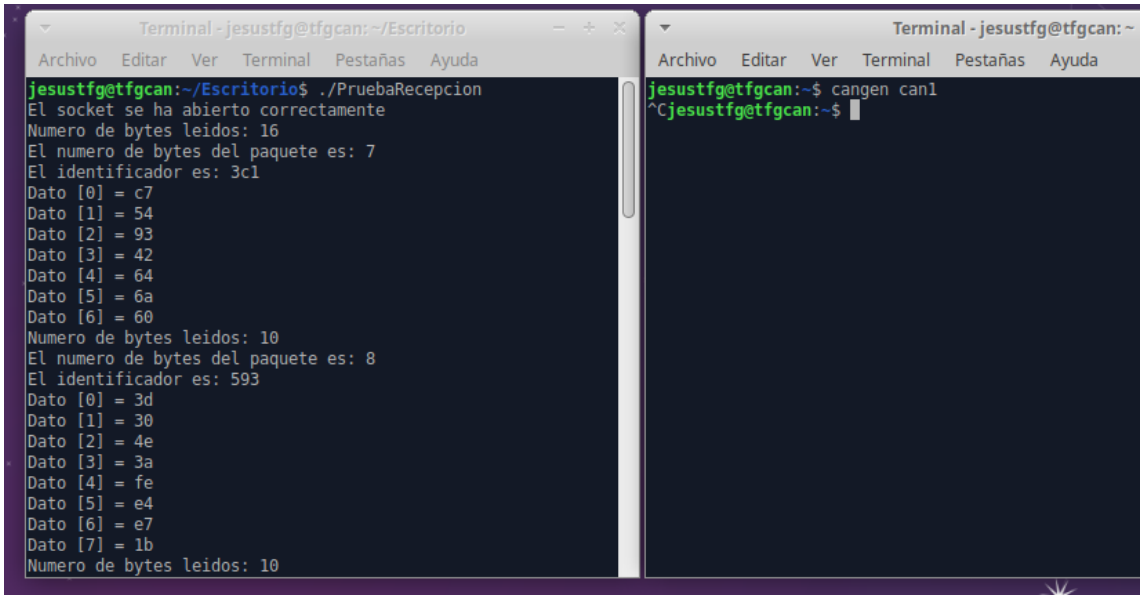

Figura 2.10 Prueba programa de recepción ejecutada en PC

2.2.4 Pruebas de envío y recepción en BeagleBone Black

Para hacer la misma prueba de envío y recepción, pero ahora en la BeagleBone Black debemos de realizar primero una serie de pasos para configurar el dispositivo.

En primer lugar, hay que conectarse a la BBB a través de ssh, al igual que se hizo en el apartado 2.1, y crear un directorio que en nuestro caso se llama “pruebas”:

- *mkdir pruebas*

y posteriormente se ejecuta el comando exit, para salir de la conexión con la BBB.

Una vez esté creado el directorio, usando el comando scp, se realizará la copia del archivo desde el PC a la BeagleBone:

- *scp ~/PRUEBAENVIO.cpp debian@192.168.7.2: ~/pruebas*

A continuación, para ejecutar el archivo, hay que volver a conectarse a la BeagleBone Black y navegar al directorio (cd pruebas)-

Por otra parte, es necesario compilar el código antes de poder ejecutarlo, por lo que se necesitará la instalación de un compilador C++:

-*sudo apt-get install g++*

y realizar la correspondiente compilación:

-*g++ -o PRUEBAENVIO PRUEBAENVIO.cpp*

creando así un archivo ejecutable llamado PRUEBAENVIO, que se ejecuta usando: ./PRUEBAENVIO.

Finalmente, es necesario conectarnos también a la BBB para configurar el controlador CAN1, como se explica en el apartado 2.1 a la espera de recibir tramas desde el CAN0.

Es importante destacar que gracias a SocketCAN, aunque se trabaje en máquinas con arquitecturas distintas, o con diferentes dispositivos CAN, es posible compilar el código sin la necesidad de realizar cambios entre una máquina y otra.

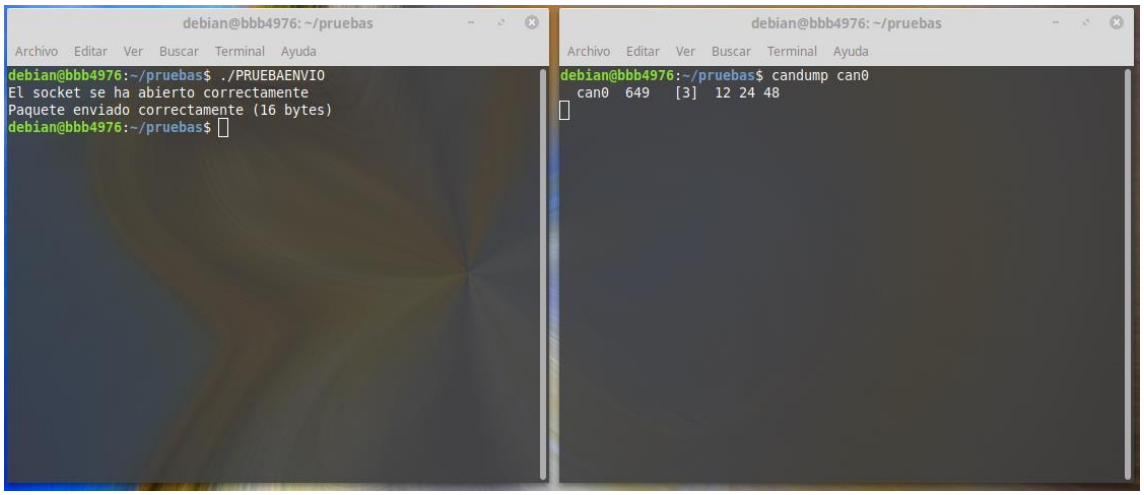

Figura 2.11 Prueba programa de envío ejecutado en BBB.

Por otra parte, para la PRUEBARECEPCION.cpp se ejecutan los mismos pasos:

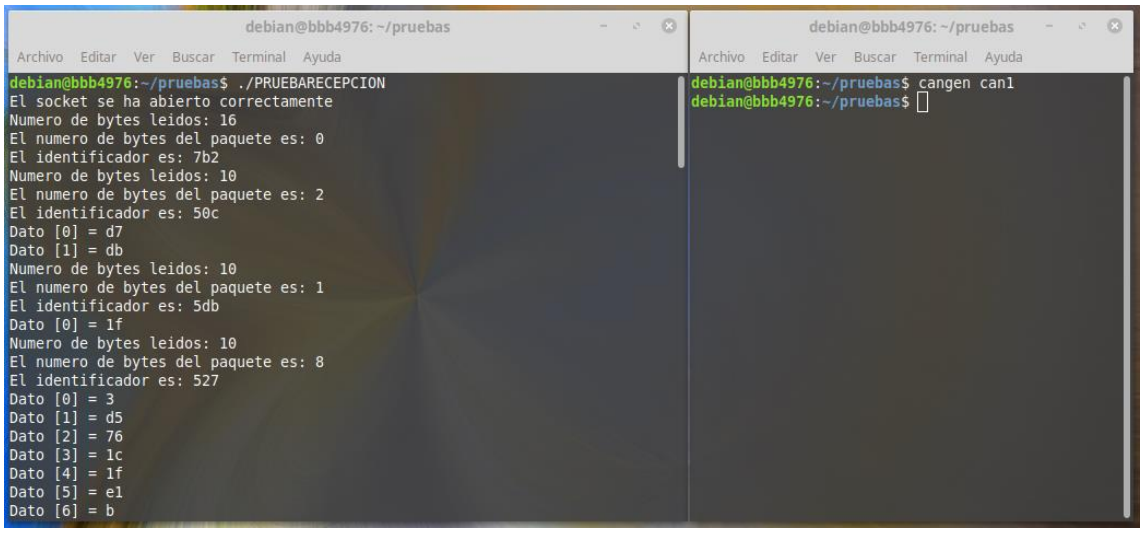

Figura 2.12 Prueba programa de recepción ejecutado en BBB.

2.3. Pruebas de envío y recepción con filtros

En el contexto de la programación de redes de comunicación, los filtros de paquetes son herramientas útiles para limitar el tráfico de red y mejorar el rendimiento de la comunicación. Estos filtros permiten a los desarrolladores implementar lógica personalizada para procesar solo los paquetes relevantes que cumplen con ciertos criterios de filtrado. Al implementar filtros de paquetes en programas de red, se mejora la eficiencia y la fiabilidad de la aplicación al reducir el consumo de recursos y minimizar la posibilidad de errores de comunicación. Antes de comenzar con el desarrollo del programa de recepción de tramas CAN mediante el uso de filtros, se va a crear un programa de envío de tramas CAN que permita al usuario introducir el identificador del mensaje por teclado. Para esto, se utilizará el programa de envío desarrollado anteriormente en el apartado 2.2.1. donde, en lugar de especificar la identificación del mensaje en el comando de envío, se permitirá al usuario introducir la identificación del mensaje mediante la entrada estándar del teclado. De esta forma, se podrá enviar un mensaje CAN personalizado en tiempo real y sin necesidad de recompilar el programa cada vez que se quiera enviar un mensaje con una identificación diferente Este programa de envío de tramas será una buena base para el desarrollo del programa de recepción de tramas mediante el uso de filtros, ya que permitirá al usuario enviar tramas con identificadores específicos que luego podrán ser filtrados en el programa de recepción para procesar solo los mensajes relevantes.

2.3.1 Programa de envío con ID personalizado

El código del mismo se corresponde con una modificación del código de envío, en el cual el valor del identificador se obtiene al ser leído desde el teclado:

```
int identificador = 0;
struct can_frame frame;
int nbytes = 0;
for (int i = 0; i < 10; i++){
    // Paquete a enviar
    std::cout << "Ingrese el valor del identificador" << std::endl;
    std::cin >> identificador;
    frame.can_id = identificador;
    frame.can_dlc = 3;
    frame.data[0] = 0x12;
    frame.data[1] = 0x24;
    frame.data[2] = 0x48;
    // Envío de paquete
    nbytes = write(s, &frame, sizeof(struct can_frame));
        if (nbytes == -1) {
            std::cout << "Error al enviar el paquete" << std::endl;
        } else {
        std::cout << "Paquete enviado correctamente (" << nbytes << " bytes)"
        << std::endl;
    }
}
```

Figura 2.13. Modificación programa de envío para ID personalizado

Este debe ser insertado después de la función “setsockopt” de la retroalimentación. Como solo interesa el hecho de enviar identificadores distintos, los datos a enviar no se modifican para la prueba.

Por lo tanto, como se explica en los apartados 2.1 y 2.2.4, una vez hallamos transferido el archivo a la BeagleBone, realizados todos los pasos necesarios, y usando el comando “candump” para recibir tramas desde CAN0, ejecutamos el código de envío con ID personalizado:

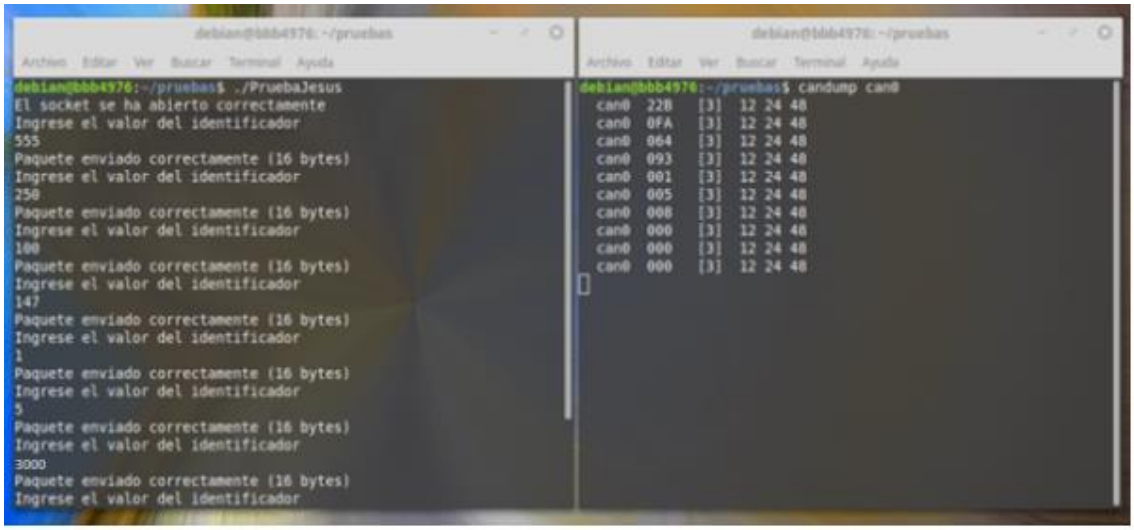

Figura 2.14. Prueba programa de envío para ID personalizado

Es importante aclarar que, al estar usando tramas estándar, el identificador tiene 11 bits y se representa en hexadecimal. El rango de identificadores permitidos en una trama CAN estándar es de 000 a 7FF en hexadecimal, lo que corresponde a un rango de 0 a 2047 en decimal. Por lo tanto, como se observa en la figura 2.14, al introducir cualquier valor fuera de rango, la trama no se enviará correctamente a través del BUS CAN. Esto hay que considerarlo a la hora de establecer los identificadores que se van a usar en el programa de recepción con filtro.

Una vez hemos realizado la prueba en el programa de envío con los identificadores introducidos por el usuario, y comprobado su correcto funcionamiento, el mismo será utilizado posteriormente en el desarrollo del proyecto.

2.3.2 Programa de recepción con filtro

Según se explica en la página web [12], se puede establecer un filtro para un identificador concreto, usando la función "setsockopt" con la opción CAN_RAW_FILTER. Esta opción permite establecer un filtro para que el socket sólo reciba tramas que cumplan ciertas condiciones.

```
//Filtros
struct can_filter rfilter [1];
rfilter[0].can_id = 0x248;
rfilter[0].can_mask = CAN_SFF_MASK;
```

Figura 2.15. Filtro ID 248

Como se observa en la figura 2.15 se establece un filtro de identificador en el socket s para permitir sólo tramas con identificador 0x248. La máscara CAN_SFF_MASK se utiliza para permitir sólo identificadores de trama estándar, es decir, de 11 bits. Esta parte de código debe ser insertado después de la llamada a la función "bind", antes del inicio del bucle de recepción de tramas del programa de recepción. Con este filtro, el programa sólo recibirá tramas que cumplan la condición del filtro, en este caso, las tramas con el identificador 0x248.

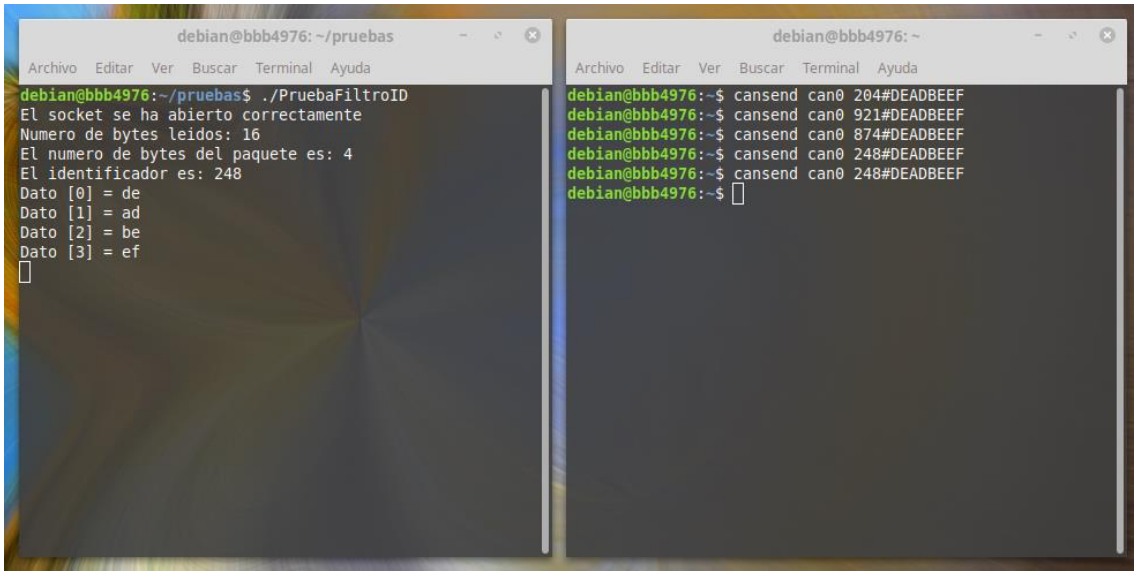

Figura 2.16 Prueba filtro ID 248

Como podemos observar en la figura 2.16 al establecer el filtro, todos los mensajes que se envíen y no tengan el identificador 248, no serán mostrados por parte de ese nodo con el filtro.

2.4. Comunicación PC – BeagleBone Black

Una vez se ha realizado la comunicación entre los controladores de cada dispositivo por separado, el siguiente paso será realizar una prueba de comunicación entre uno de los controladores del PC y de la BeagleBoneBlack. Para ello lo primero que hay que hacer es conectar físicamente los dos controladores, conectando los cables CAN HIGH y CAN LOW que interconectan los dos transceptores de la BeagleBone como se observa en la figura 2.1, con los cables CAN HIGH y CAN LOW del PC. Dicha configuración se muestra en la figura 2.17:

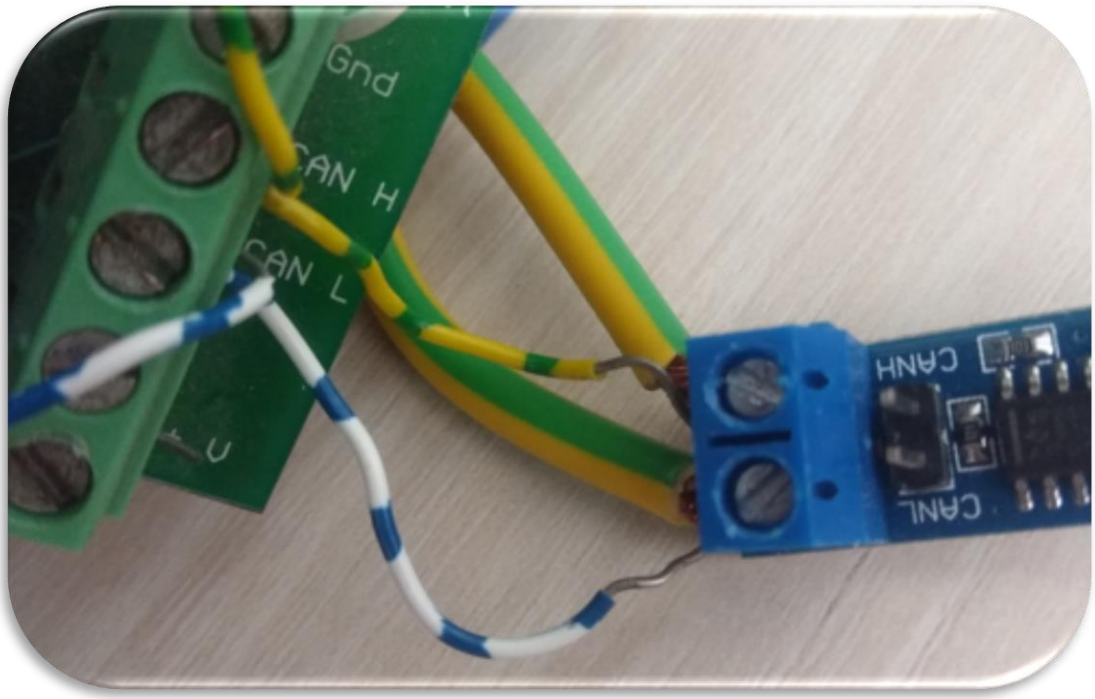

Figura 2.17 Conexión física PC-BBB

Figura 2.18 Conexión física PC-BBB con RJ-45

En la figura 2.18 vemos como una vez hemos conectado el CAN HIGH y el CAN LOW, la conexión con los controladores del PC se lleva a cabo con la conexión de un cable de conexión tipo RJ-45 en el extremo de la conexión con la BBB, y en el otro extremo con un conector DB9. En este caso se llevará a cabo una prueba de envío y recepción de datos a través del bus CAN utilizando las pruebas de envíos y recepción de datos creadas en el apartado 2.2. Para ello es necesario configurar antes los dispositivos, en el caso de la BBB, es necesario que se realice la configuración que se explica en el apartado 2.1 en profundidad, donde se establece la velocidad de transmisión de los datos, se configura el levantamiento del controlador, y se configuran los pines para usarse tipo CAN:

- *sudo ip link set can0 up type can bitrate 125000*
- *sudo ifconfig can0 up*
- *config pin P9_19 can*
- *config pin P9_20 can*

En el caso del PC, es necesario el *bitrate* y el levantamiento únicamente:

- *sudo ip link set can1 up type can bitrate 125000*

- sudo ifconfig can1 up

Como podemos observar, para la BeagleBone Black se ha llevado a cabo la configuración del controlador CAN0, y para el PC el controlador CAN1, aunque de igual modo si se activa el CAN0 en los dos, sería lo mismo, únicamente se ha hecho para distinguir cual es el del PC y cual el de la BBB.

Una vez se ha completado el proceso de configuración, utilizando las pruebas de envío y recepción se ha realizado la correspondiente prueba con éxito, como se observa en las figuras 2.19 y 2.20. En la BBB se ha ejecutado el código de envío, habiendo transferido y compilado previamente el archivo en el apartado 2.2.4, y en el PC se ha ejecutado el código de recepción

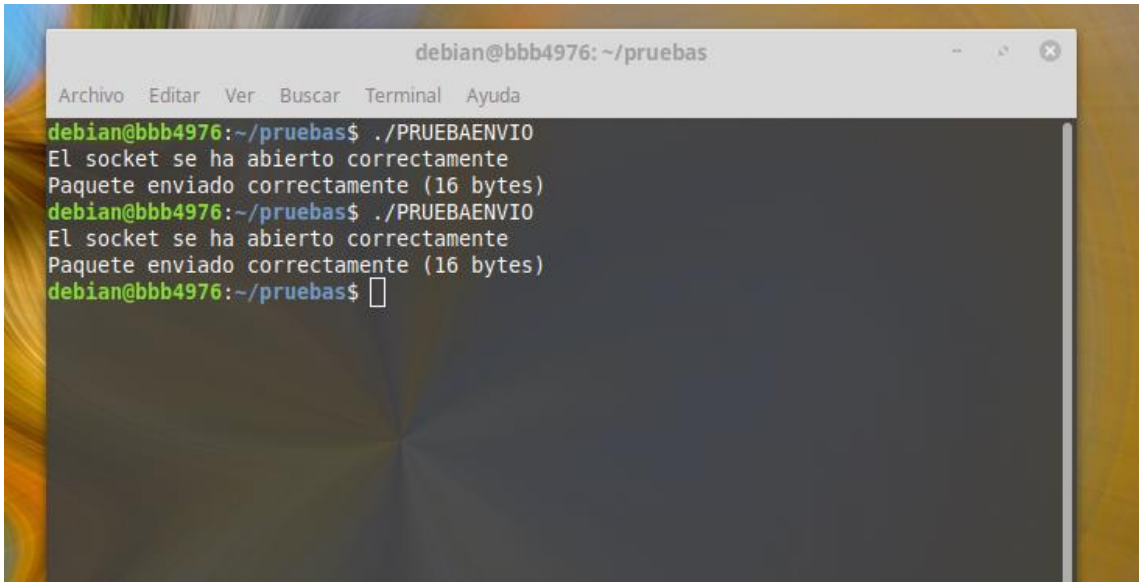

Figura 2.19 Prueba envío PC-BBB

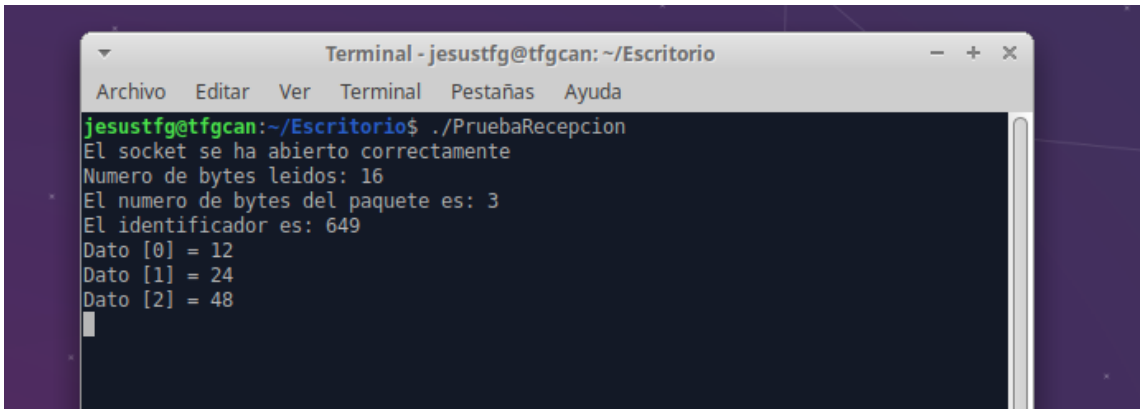

Figura 2.20 Prueba recepción PC-BBB

2.4.1 Programa de control para filtros

Como ya se ha hablado del uso de filtros, el siguiente paso que se llevará a cabo será el desarrollo de un programa que implementa una acción de control para un determinado identificador, y responde enviando una trama por el bus. Para ello se llevará a cabo la siguiente modificación en el código de recepción, al cual le habíamos aplicado un filtro.

```
//Filtro
struct can_filter rfilter [1];
rfilter[0].can_id = 0x248;
rfilter[0].can_mask = CAN_SFF_MASK;


setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));

//Trama de control para ID 248
struct can_frame control;
control.can_id = 0x741;
control.can_dlc = 4;
control.data[0] = 0x51;
control.data[1] = 0x92;
control.data[2] = 0x34;
control.data[3] = 0x67;
```

Figura 2.21. Definición de la estructura de control

Una vez se había declarado el filtro, el siguiente paso consiste en declarar una trama de control que se envíe por el bus cuando se reciban tramas con el identificador del filtro. Por lo tanto, luego cuando se reciban las tramas, dentro del else del "paranoid check", justo después del bucle for que muestra los datos de la trama recibida, como se observa en figura 2.8, se debe colocar la función "write", pero esta vez con la referencia a la estructura control:

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

De este modo, cuando se reciba la trama y se muestre, se enviará también la trama que hemos definido de control. Para llevar a cabo la ejecución de esta prueba, se ejecutará la prueba desde la BeagleBone Black, y desde el PC se enviará la trama de control, además de recibirse la correspondiente trama de control.

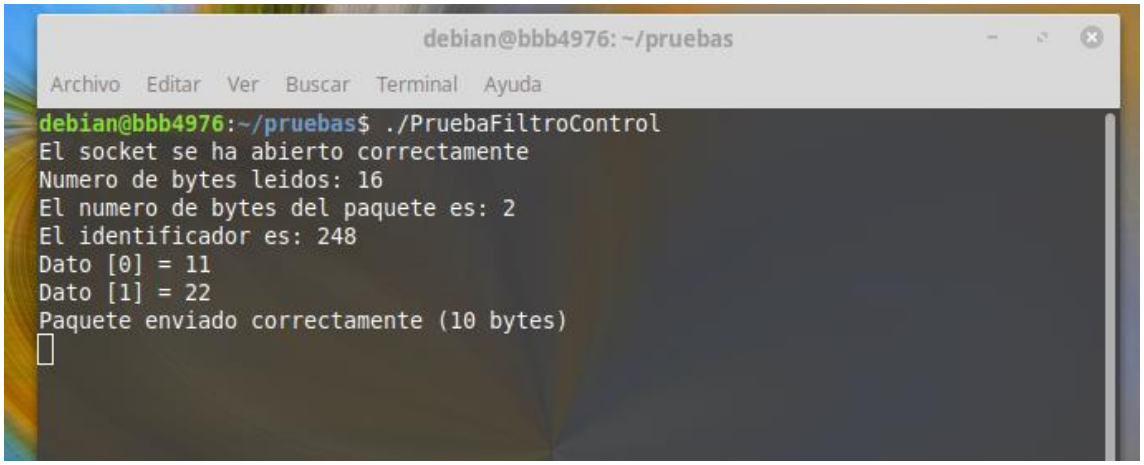

Figura 2.22. Ejecución Prueba filtro con control

En la figura 2.22 se observa cómo se ha recibido la trama en la BBB, con el identificador 0x248 y se ha mostrado. El controlador está asociado al CAN0, y por lo tanto en la figura 2.23 se muestra la recepción de la trama de control por el CAN1, además del envío de la trama cuyo identificador es el filtro.

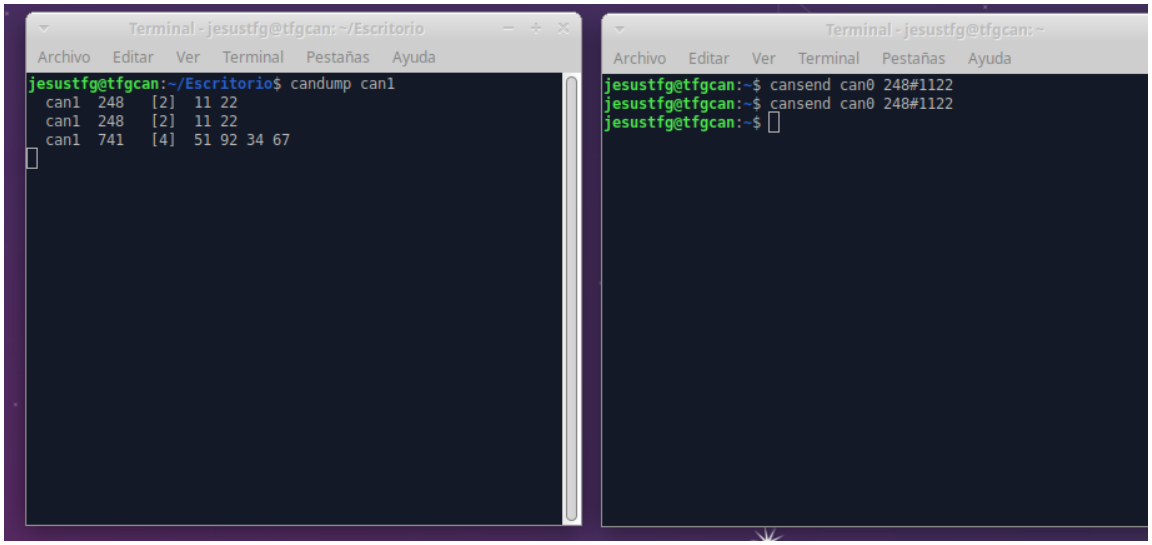

Figura 2.23 Envío trama filtro y recepción trama control

2.4.2 Programa de control directo para filtros

Pero la acción de control anterior se puede ejercer directamente sobre la trama que se recibe en el bus, es decir, que no se vuelva a enviar la trama recibida, y ejercer una acción a modo de actuación (enviar la trama de control), sino que directamente se modifique la trama recibida y se ejerza alguna modificación sobre

la misma. Por lo tanto, una vez hemos declarado el filtro, en el bucle de recepción de tramas, se debe realizar la siguiente modificación:

```
    /* paranoid check... */
    while(1){
        nbytes = 0;

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

        if((nbytes < 0) || ((unsigned) nbytes < sizeof (struct can_frame))){
            std::cerr << "La trama esta vacia o incompleta" << std::endl;
            exit(0);
        }
        else{
             for(int i = 0; i < frame.can_dlc; i++){
                frame.data[i] += 4;
            }



            std::cerr << "Numero de bytes leidos: " << nbytes << std::endl;

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

            std::cout << "El identificador es: " << std::hex
                << 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;
            }
            write ( s, &frame, sizeof (struct can_frame));
        }

    }

    return 0;
}
```

Figura 2.24. Modificación de bucle de recepción para control directo

Como se puede observar en la figura 2.24, la trama que se recibe, y que cumpla con el filtro de identificador 0x248, será mostrada pero esta vez se ejerce una acción que modifica directamente los datos de la trama recibida. Al igual que antes, justo después del bucle for que muestra los datos de la trama recibida, se debe colocar la función “write”, si se quiere volver a enviar la trama recibida.

Por lo tanto, si ahora establecemos conexión entre la BBB y el PC, y desde el PC enviamos la siguiente trama:

- *cansend can1 248#23 34 12*

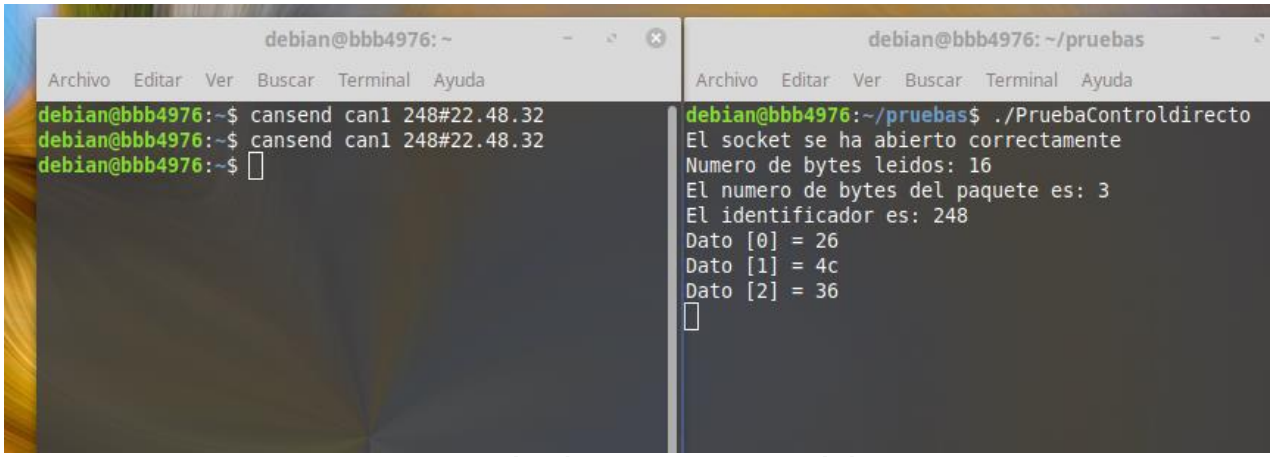

Figura 2.25. Prueba de trama con control directo

En la BeagleBone Black se recibe la trama con el identificador 0x248, pero se cambia el valor de los datos, sumándole 4 unidades. La capacidad de modificar y reenviar tramas recibidas puede ser muy útil para depurar problemas de red, filtrar tramas no deseadas, así como simular dispositivos o realizar pruebas de seguridad.

2.5. Clases para gestionar la comunicación

Después de realizar pruebas de comunicación con la API SocketCAN de envío y recepción de tramas CAN mediante programas desarrollados en C++, se ha decidido crear unas clases para facilitar la creación de nuevos programas. Con estas nuevas clases, se podrá ahorrar tiempo y esfuerzo en la implementación de los programas, ya que se tendrá acceso a una serie de herramientas que permitirán trabajar de manera más eficiente y cómoda. Además, las clases proporcionarán una interfaz sencilla y clara que simplificará el proceso de diseño y depuración de los programas, lo que se traducirá en un menor tiempo de desarrollo y un mayor control sobre el código. Entre las principales ventajas que se esperan obtener con la utilización de estas clases se incluyen una mayor flexibilidad, una mejor organización del código, una mayor legibilidad y facilidad para realizar modificaciones y mejoras en el futuro.

2.5.1 Clase CAN

La clase CAN tiene como objetivo principal crear y gestionar sockets de comunicación CAN en Linux. En la primera sección del código, se incluyen las librerías necesarias para poder utilizar las funciones y estructuras necesarias para manipular los sockets CAN en Linux. El constructor de la clase CAN es el encargado de crear el socket y asociarlo con el controlador CAN indicado. Recibe como parámetro el número del controlador a usar (0 o 1) y lo asocia con el socket. Además posee dos métodos, el método getCAN, que simplemente devuelve el nombre del controlador que se ha asociado al socket, y el método getSocket que devuelve el valor del socket que se ha creado.

```
#ifndef _CAN_
#define _CAN_

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <linux/can.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <linux/can/raw.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <unistd.h>


class CAN {


private:
    // Declaración del socket
    int s;
    // Declaración de la estructura de un socket CAN
    struct sockaddr_can addr;
    // Declaración de estructura de red
    struct ifreq ifr;
    // Nombre del controladror can asociado
    std::string nombre_can;

public:
    // Constructor
    CAN(int c);
    // Destructor
    ~CAN();
    // Devuelve el valor del controlador can asociado
    std::string getCAN() const;
    // Devuelve el valor del socket
    int getSocket();
    // Decir que la clase FRAME es amiga
    friend class Frame;
};

#endif
```

Figura 2.26. Clase CAN .hpp

```cpp
#include "CAN.hpp"
#include <iostream>

//Constructor
CAN::CAN(int c) {

    while(true){
        //Asociamos el controlador
        if (c == 0 || c == 1) {
            break;
        }
        std::cerr << "Introduzca un controlador correcto ( 0 o 1 )" << std::endl;
    }

    this->nombre_can = "can" + std::to_string(c);

    // Abrimos el socket
    this->s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

    // Comprobar que se abrió bien
    if (this->s == -1) {
        std::cerr << "El socket no se ha abierto correctamente" << std::endl;
        exit(1);
    } else {
        std::cout << "El socket se ha abierto correctamente" << std::endl;
    }

    // Unir socket a dirección y puerto local
    strcpy(this->ifr.ifr_name, this->nombre_can.c_str());
    ioctl(this->s, SIOCGIFINDEX, &ifr);

    this->addr.can_family = AF_CAN;
    this->addr.can_ifindex = this->ifr.ifr_ifindex;

    if (bind(this->s, (struct sockaddr *)&addr, sizeof(this->addr)) == -1) {
        std::cerr << "No se pudo asociar el socket al controlador CAN" << std::endl;
        exit(1);
    }

    // Retroalimentación
    int loopback = 0;
    setsockopt(this->s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
}

//Destructor
CAN::~CAN() {
    close(this->s);
}

//Método que muestra por pantalla el controlador usado
std::string CAN::getCAN() const {
    return this->nombre_can;
}

int CAN::getSocket(){
    return this->s;
}
```

Figura 2.27. Clase CAN .cpp

En la figura 2.27, se puede ver como primero, se define el constructor, que recibe un argumento c que indica el número del controlador CAN que se desea asociar al socket. Dentro del constructor, se realiza un ciclo que espera a que se ingrese un valor correcto para c (0 o 1). Luego, se construye el nombre del controlador y se abre el socket mediante la función socket de la librería sys/socket.h. Si el socket se abre correctamente, se unen la dirección y el puerto local mediante la función bind, es decir se asocia la estructura "sockaddr_can" con el socket s, y por tanto el socket ya estaría listo para enviar y recibir tramas desde la interfaz especificada en "can_ifindex". Si esta unión no se puede realizar, el programa finaliza con un mensaje de error. Por último, se configura el socket para que no se use la función de loopback y se define el destructor, que simplemente cierra el socket. También como ya se mencionó se definen dos métodos adicionales: getCAN devuelve el nombre del controlador usado, y getSocket devuelve el valor del socket s.

2.5.2 Clase Frame

La clase Frame tiene como funcionalidad crear, almacenar, obtener y enviar tramas CAN, utilizando la estructura "can_frame". Para ello, la clase cuenta con los siguientes métodos:

- Frame: constructor que inicializa la trama a 0.
- createFrame(uint16_t id, const std::string& data_hex): método que crea una trama a partir de un identificador de 11 bits y un string que contiene los datos en formato hexadecimal. La longitud de los datos no puede ser mayor a 8 bytes.
- getId const: método que devuelve el identificador de la trama.
- getcan_dlc const: método que devuelve la longitud de los datos.
- getdata const: método que devuelve los datos de la trama en formato string hexadecimal
- enviarTrama: método que envía la trama a través del socket CAN.
- recibirTrama: método que lee una trama desde el socket CAN y muestra su contenido por la consola. En caso de no haber tramas disponibles, se queda en espera hasta recibir una.

```
#ifndef _FRAME_
#define _FRAME_


#include <vector>
#include <string>
#include <cstdint>
#include "CAN.hpp"

class Frame {

private:
    struct can_frame frame;
    //Obtener el socket de la clase CAN
    const int socket  = CAN::getSocket();
public:
    // Constructor, inicializa la trama a 0
    Frame();
    // Método para crear una trama
    void createFrame(uint16_t id, const std::string& data_hex);
    // Métodos para obtener los atributos
    uint16_t getId() const;
    uint8_t getcan_dlc() const;
    std::string getdata() const;
    //Metodo para enviar trama
    void enviarTrama();
    //Metodo para recibir tramas
    void recibirTrama();

};

#endif
```

Figura 2.28. Clase Frame.hpp

La clase tiene un miembro privado “frame” que es una estructura can_frame que almacena los datos de la trama. La estructura can_frame es un tipo de dato definido en la biblioteca de sockets CAN para Linux, que contiene información sobre una trama CAN, como su identificador y datos. En la parte publica tenemos:

• El constructor, que es el primer método que se llama cuando se crea una instancia de la clase Frame. En este caso, el constructor inicializa la estructura “can_frame” a 0.

• El siguiente método es “createFrame”, que se utiliza para crear una trama CAN con un ID de 11 bits y datos hexadecimales. Primero, la función comprueba que el ID sea válido (máximo 11 bits) y luego convierte los datos hexadecimales en un vector de uint8_t. A continuación, la función verifica que la longitud de los datos no sea mayor de 8 bytes y llena la estructura can_frame con el ID y los datos.

• El siguiente método, getId, simplemente devuelve el identificador de la trama.

• El método getcan_dlc devuelve la longitud de los datos en la trama, que se almacena en el campo can_dlc de la estructura can_frame.

• El método getdata convierte los datos hexadecimales almacenados en la estructura can_frame en una cadena de caracteres. Primero, se utiliza un stringstream para construir la cadena de caracteres y se establece el relleno a '0'. Luego, para cada byte de datos en la trama, el método convierte el byte hexadecimal en un número entero y lo agrega al stringstream como un número entero de dos dígitos. Finalmente, el método devuelve la cadena de caracteres.

• El método enviarTrama es utilizado para enviar la trama a través del socket de la clase CAN. El método simplemente llama a la función write de la biblioteca de sockets de UNIX y pasa la estructura can_frame como argumento.

• Por último, el método recibirTrama es utilizado para recibir tramas CAN a través del socket de la clase CAN. Este método utiliza la función read de la biblioteca de sockets de UNIX para leer los datos del socket. Si se leen correctamente, el método muestra en la consola el número de bytes leídos, el identificador de la trama y los datos hexadecimales de la trama.

El método enviarTrama y recibirTrama de la clase Frame necesitan tener acceso al socket para poder enviar y recibir tramas, respectivamente. Sin embargo, el socket es una variable privada de la clase CAN y no es accesible desde la clase Frame. Para solucionar este problema, se ha utilizado la relación de amistad. Al declarar getSocket como una función amiga de la clase Frame, se permite que la función getSocket tenga acceso a las variables privadas de la clase CAN. De esta manera, cuando la función getSocket es llamada dentro de la clase Frame, puede obtener el valor del socket y utilizarlo para enviar o recibir tramas. También se declara la variable s de la clase Can como estática para que pueda ser accedida por la clase Frame sin necesidad de crear una instancia de la clase Can. Al declarar la variable como estática, se reserva un solo espacio de memoria para la variable s en lugar de reservar uno para cada instancia de la clase Can. Esto significa que cuando se modifica la variable s en una instancia de la clase Can, el cambio se refleja en todas las instancias de la clase Can y en la clase Frame.

```cpp
#include <iostream>
#include <iomanip> // para usar std::hex y std::setw
#include "Frame.hpp"
#include <vector>

//Constructor
Frame::Frame(){
//Inicializar la trama a 0
this->frame.can_id = 0;
this->frame.can_dlc = 0;
memset(this->frame.data, 0, sizeof(this->frame.data));
}

//Crear Trama
void Frame::createFrame(uint16_t id, const std::string& data_hex) {
    // Validar que el id sea válido (máximo 11 bits)
    if (id > 0x7FF) {
        std::cerr << "Error: el id debe ser máximo de 11 bits" << std::endl;
        exit(1);
    }

    // Convertir el string de datos hexadecimales a un vector de uint8_t
    std::vector<uint8_t> datos;
    for (size_t i = 0; i < data_hex.length(); i += 2) {
        std::string byteString = data_hex.substr(i, 2);
        uint8_t byte = (uint8_t)std::stoul(byteString, nullptr, 16);
        datos.push_back(byte);
    }

    // Validar que la longitud de los datos no sea mayor a 8 bytes
    if (datos.size() > 8) {
        std::cerr << "Error: la longitud de los datos no puede ser mayor a 8 bytes"
                  << std::endl;
        exit(1);
    }

    // Llenar la estructura can_frame
    this->frame.can_id = id;
    this->frame.can_dlc = (uint8_t)datos.size();
    for (size_t i = 0; i < this->frame.can_dlc; i++) {
        this->frame.data[i] = datos[i];
    }
}

// Obtener identificador de la trama
uint16_t Frame::getId() const {
    return this->frame.can_id;
}

// Obtener longitud de los datos
uint8_t Frame::getcan_dlc() const {
    return this->frame.can_dlc;
}

// Obtener datos
std::string Frame::getdata() const {
    std::stringstream tramastring;
    tramastring << std::hex << std::setfill('0');
    for (size_t i = 0; i < this->frame.can_dlc; i++) {
        //Transforma los datos, cada par hexadecimal almacenado en binario
        //a cadena de caracteres
        tramastring << std::setw(2) << (int)frame.data[i];
    }
    return tramastring.str();
}
```

```
// Envío de paquete
void Frame::enviarTrama(){
    write(this->socket, &frame, sizeof(struct can_frame));
}

//Recepción de paquetes
void Frame::recibirTrama(){

ssize_t nbytes = read(this->socket, &frame, sizeof(struct can_frame));

    /* paranoid check... */
    while(1){
        nbytes = 0;

        while (nbytes == 0){
            nbytes = read(this->socket, &frame, sizeof(struct can_frame));
        }

        if((nbytes < 0) || ((unsigned) 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;
            }
        }
    }

}
```

Figura 2.29 Clase Frame.cpp

Analizando en profundidad la función “createFrame”, La máscara del ID utilizada en este caso, 7FF en hexadecimal o 2047 en decimal, se debe al hecho de que se está utilizando el estándar CAN 2.0A, y por lo tanto solo serán aceptados los ID que sean menores que el valor 7FF, es decir, 11 bits.

Luego, se convierte el string que representa datos en hexadecimal a un vector de bytes de tipo uint8_t. Es una operación común en la programación de comunicación de datos, especialmente en el contexto de CAN, donde los mensajes se transmiten en formato binario. Primero, el código recorre el string de datos hexadecimales en incrementos de dos. Dentro del bucle for, la función “substr” extrae dos caracteres hexadecimales adyacentes del string, creando una subcadena de dos caracteres que representa un byte en hexadecimal. Luego, la función stoul convierte la subcadena de dos caracteres en un entero sin signo de 8 bits de tipo

uint8_t en base hexadecimal. El byte convertido se agrega al vector datos mediante la función push_back.

Por otra parte, la función “getData” se utiliza para obtener los datos de la trama en formato de cadena de caracteres. Para hacer esto, el método recorre los datos de la trama utilizando un bucle for que se ejecuta una vez por cada byte en la trama (la longitud de los datos de la trama se almacena en el campo can_dlc de la estructura can_frame). En cada iteración del bucle, el método utiliza la función setw() para asegurarse de que cada byte se formatee como un par de caracteres hexadecimales de dos dígitos. Luego, el byte se convierte en un entero y se agrega a un stringstream utilizando el operador de inserción “<<”. Al final del bucle, el método devuelve la cadena de caracteres completa utilizando la función str() del stringstream. Además, se utiliza el tipo de dato size_t, que se usa comúnmente en C++ para representar tamaños de objetos o matrices, y también se utiliza para iterar a través de ellos. Es un tipo de datos sin signo y su tamaño está garantizado para ser lo suficientemente grande como para representar la cantidad máxima de memoria que se puede asignar en un sistema. Al utilizar size_t en lugar de tipos de datos firmados como int o long, se evitan problemas con valores negativos, ya que no tiene sentido tener un tamaño de objeto o matriz negativo. Además, los tamaños de objetos y matrices son siempre no negativos, por lo que utilizar un tipo de datos sin signo es más apropiado y reduce la posibilidad de errores de programación.

2.5.3 Prueba clase CAN y Frame

La siguiente prueba crea un objeto de la clase CAN, eligiendo el controlador “can0”, y un objeto de la clase Frame. Luego se llama al método createFrame del objeto Frame para crear una trama con ID 0x100 y datos "30485127". Se imprime por pantalla el nombre de la interfaz CAN asociada al objeto CAN creado y se muestran los atributos de la trama creada, como su ID, longitud de los datos y los datos en formato hexadecimal. Después se llama al método enviarTrama de la clase Frame para enviar la trama creada, y finalmente se llama al método recibirTrama del objeto recibir de la clase Frame para recibir tramas.

```cpp
#include "CAN.hpp"
#include "Frame.hpp"
#include <iostream>


int main() {
    //Crear un objeto de la clase CAN
    CAN can(0);
    // Crear un objeto de la clase Trama
    Frame trama;
    //Crear una trama
    trama.createFrame(0x100, "30485127");
    // Obtener el nombre de la interfaz CAN
    std::cout << "El controlador asociado es: " << can.getCAN() << std::endl;

    // Mostrar por pantalla los atributos de la trama creada
    std::cout << "ID de la trama: " << std::hex << trama.getId() << std::endl;
    std::cout << "Longitud de los datos de la trama: " << (int)trama.getcan_dlc()
              << std::endl;
    std::cout << "Datos de la trama en formato hexadecimal: " << trama.getdata()
              << std::endl;

    //Enviar trama
    trama.enviarTrama();

    //Recibir tramas
    Frame recibir;
    recibir.recibirTrama();

    return 0;
}
```

Figura 2.30. Código prueba clase CAN y Frame.cpp

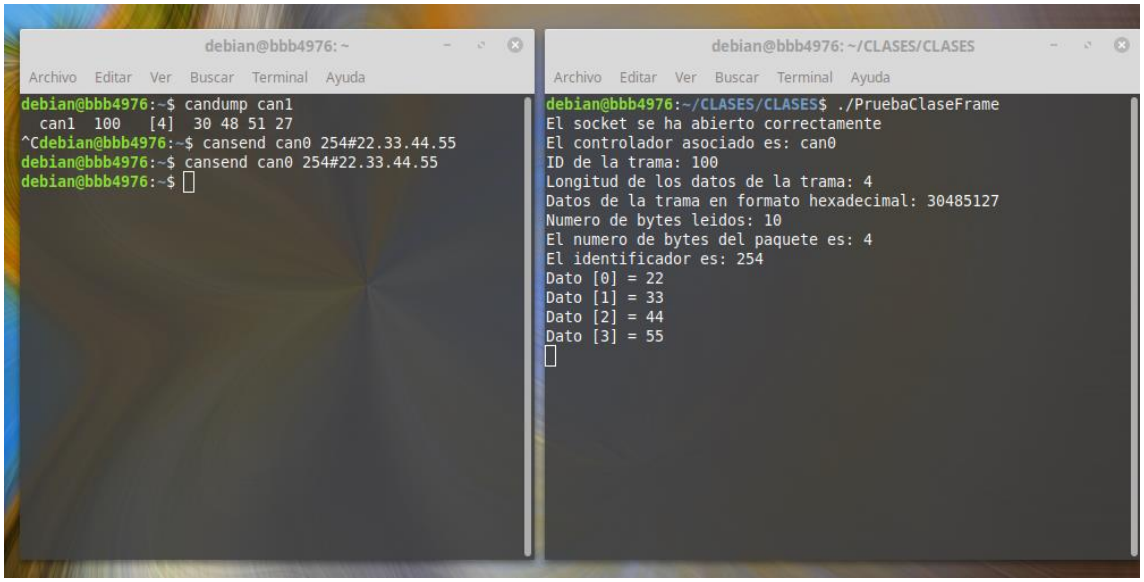

Figura 2.31. Ejecución prueba clase CAN y Frame.cpp

Por lo tanto, una vez configurada la conexión y estableciendo los comandos necesarios para realizar la comunicación entre las interfaces can0 y can1 de la BeagleBone Black, en la figura 2.31 se puede observar como una vez se ejecuta la prueba en el terminal de la derecha, se recibe en el can1 la trama definida en la prueba. Posteriormente se queda a la espera de recibir tramas, y que cuando se envía la trama con el identificador 254 hacia el can0, es recibida y mostrada en el mismo.

2.6. Clases para usar puertos analógicos, digitales y PWM de BBB

En el proyecto se van a utilizar clases desarrolladas por el tutor de este Trabajo de Fin de Grado, Alberto F. Hamilton Castro, para controlar las entradas y salidas digitales, entrada analógica, y salida PWM de la BeagleBone Black. Estas clases permiten acceder a los pines de la BBB de manera sencilla y eficiente, gracias a que han sido desarrolladas en C++. Con estas clases, se podrá leer y escribir en pines digitales, obtener lecturas analógicas y generar señales PWM, todo ello de manera fácil e intuitiva. Gracias a la implementación de estas clases, el desarrollo de la aplicación que controlará los sensores y actuadores será más sencillo.

2.6.1 Clase para entradas analógicas

La clase BBB_AD se encarga de controlar las entradas analógicas de la BeagleBone Black (BBB). En su constructor, se especifica el número de la entrada analógica que se quiere controlar (entre 0 y 7). La clase proporciona tres métodos públicos: "read", "readPer1" y "readV". El método "read" devuelve el valor del conversor analógico-digital (ADC) de la BBB, que está en el rango de 0 a 4095. Los métodos "readPer1" y "readV" devuelven el valor leído en tanto por uno y en voltios, siendo el rango de este entre 0 y 1,8 V.

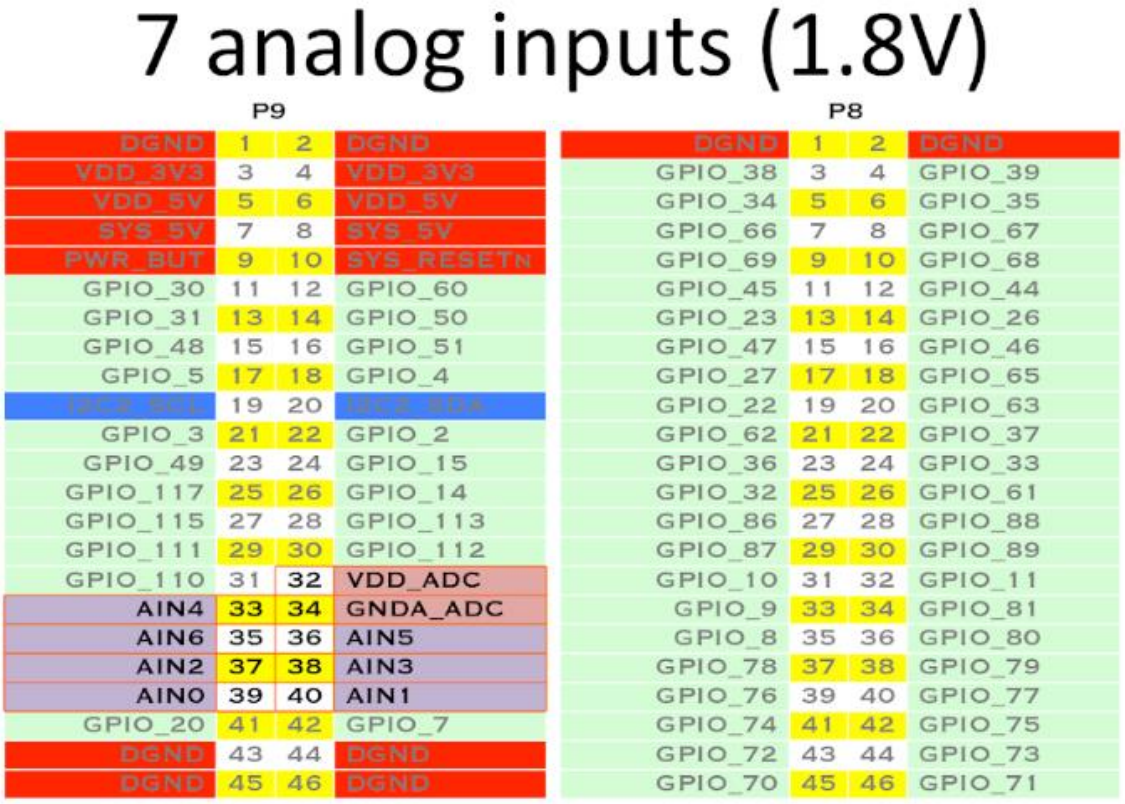

Figura 2.32. Pines de entradas analógicas en BBB

En la figura 2.32 se observa que las entradas disponibles son 7, numeradas como AIN0, AIN1, AIN2… además de disponer de una fuente de tensión en el pin 32, fijada en 1,8V, y tierra en el pin 34.

2.6.2 Prueba BBB_AD

A continuación, se pretende utilizar la clase BBB_AD junto con las clases CAN y Frame para llevar a cabo una prueba en la que se pueda leer el valor de voltaje de un potenciómetro conectado a una entrada analógica de la BBB, y luego traducir ese valor a tramas CAN que se enviarán a través del bus. La clase BBB_AD se encargará de leer el valor del potenciómetro en voltios y convertirlo en un valor entero que se usará para crear una trama CAN usando la clase Frame. Posteriormente, la clase CAN se encargará de enviar la trama por el bus. De esta manera, se podrá simular la transmisión de datos analógicos por medio del bus CAN.

```cpp
#include <iostream>
#include <iomanip> // para usar std::hex y std::setw
#include <sstream>
#include <string>
#include <unistd.h> // para usar la función sleep()
#include "Frame.hpp"
#include "BBB_AD.hpp"
#include "CAN.hpp"

int main() {

//Crear objeto para ADC
    BBB_AD a0(0);

//Crear objeto para elegir controlador
    CAN can(0);

//Crear objeto para enviar trama
    Frame trama;

//Bucle para enviar tramas cada segundo
    while(true){

    // Obtener el valor raw del ADC
        int valorRaw = a0.read();

    // Obtener los valores de readPer1 y V
        int valorPer1 = a0.readPer1() * 100;

    // Crear la cadena de caracteres que representa los datos en hexadecimal
        std::stringstream datosHex;
        datosHex << std::setfill('0') << std::setw(4) << std::hex << valorRaw
                 << std::setw(2) << std::hex << valorPer1;

        std::string datos = datosHex.str();

    //  Enviar trama
        trama.createFrame(0x100, datos);
        trama.enviarTrama();

    //  Mostrar los datos analógicos leidos
        std::cout << "Lectura Analógica raw: " << a0.read() << std::endl;
        std::cout << "Lectura Analógica %1: " << a0.readPer1() << std::endl;
        std::cout << "Lectura Analógica V: " << a0.readV() << std::endl;

    //  Esperar un segundo antes de enviar la siguiente trama
        sleep(1);
     }

  return 0;
}
```

Figura 2.33 Código prueba BBB_AD

La clase BBB_AD lee un valor analógico de la entrada A0 de la BBB y se almacena en "valorRaw" (0 a 4095). Debajo almacena el valor en tanto por ciento en la variable "valorPer1". Luego, convierte el valor en formato hexadecimal y lo envía como una trama CAN utilizando las clases Frame y CAN. El bucle principal

del programa se repite cada segundo, por lo que envía una nueva trama cada segundo. Además, también muestra los valores leídos en formato crudo, %1 y en voltios.

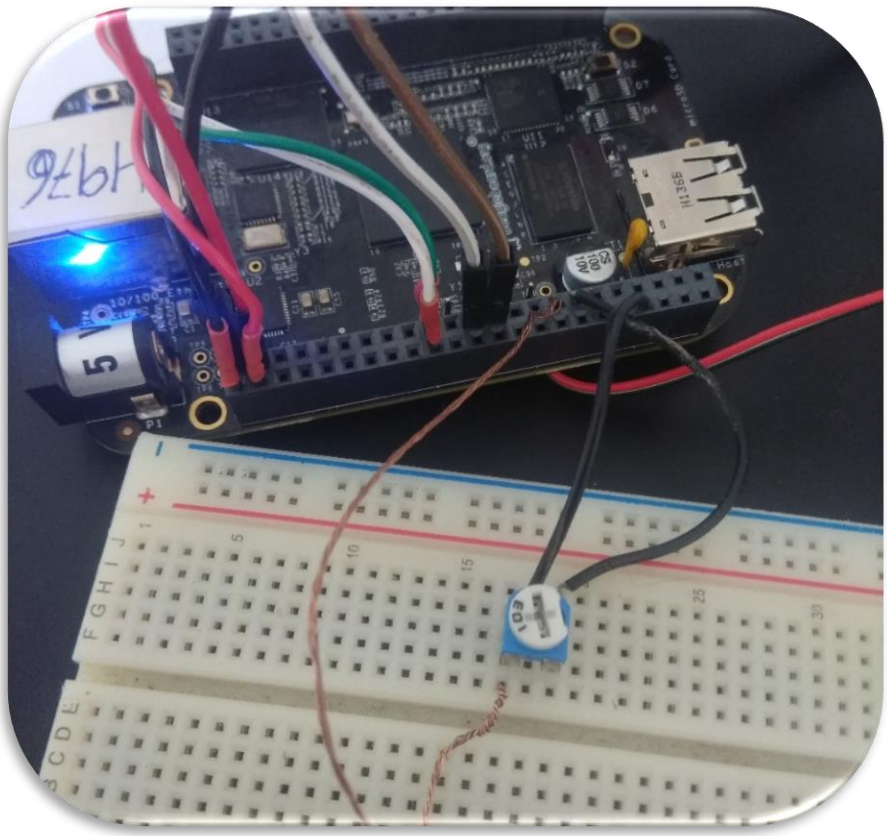

Figura 2.34. Esquema de conexión con potenciómetro

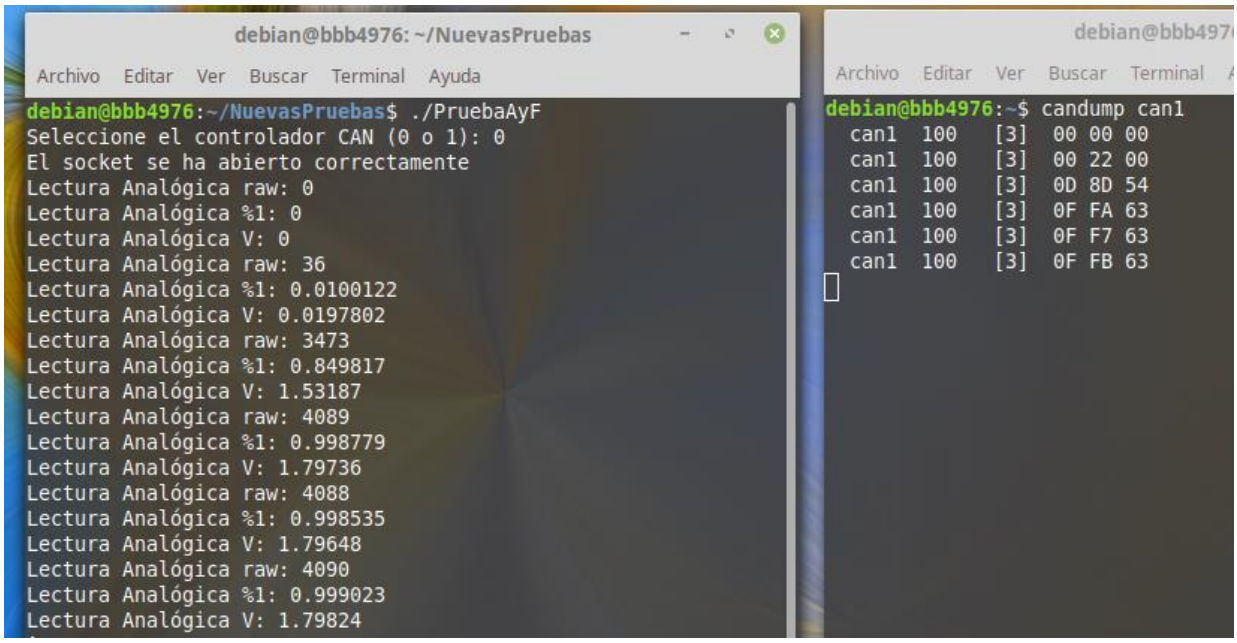

Figura 2.35. Ejecución prueba BBB_AD

En la imagen 2.34 se puede observar como un extremo del potenciómetro se conecta al pin P9-32, y el otro extremo al P9-34, estableciendo así una diferencia de potencial de 1,8V en los extremos del potenciómetro. La patilla central se conecta al pin P9-39, es decir la entrada analógica 0, que es la que se ha usado en el código para medir el voltaje, mientras se va rotando el potenciómetro.

En la imagen 6.4 se observa cómo, una vez se asocia la prueba a la interfaz can0, se transmiten los datos por el bus en formato hexadecimal, siendo el último par de dígitos el valor en tanto por ciento del voltaje, y los 2 pares que restan el valor en modo raw.

2.6.1 Clase para salida PWM

La clase BBB_PWM es una clase en C++ que controla una señal PWM (Modulación por Ancho de Pulso) en la placa de desarrollo BeagleBone Black (BBB). En este trabajo, se usará el método "setDutyPer1" de la clase BBB_PWM debido a la necesidad de establecer el ciclo de trabajo en un rango entre el 0% y el 100% (representado como valores entre 0.0 y 1.0). Esta funcionalidad es crucial para el sistema de control que se implementará posteriormente. Al utilizar setDutyPer1, se podrá ajustar el ciclo de trabajo de manera precisa y proporcional, lo que me permitirá controlar dispositivos o realizar acciones específicas en función de las necesidades del sistema de control. Esto asegurará un control preciso y efectivo sobre los componentes controlados por el PWM, brindando la flexibilidad requerida para lograr los objetivos del sistema de control en el trabajo.

La prueba que se llevará acabo utilizado esta clase, se llevará a cabo en el sistema de control que se muestra a continuación. Además en la figura 2.36 se muestran los distintos pines que se pueden usar como salida PWM.

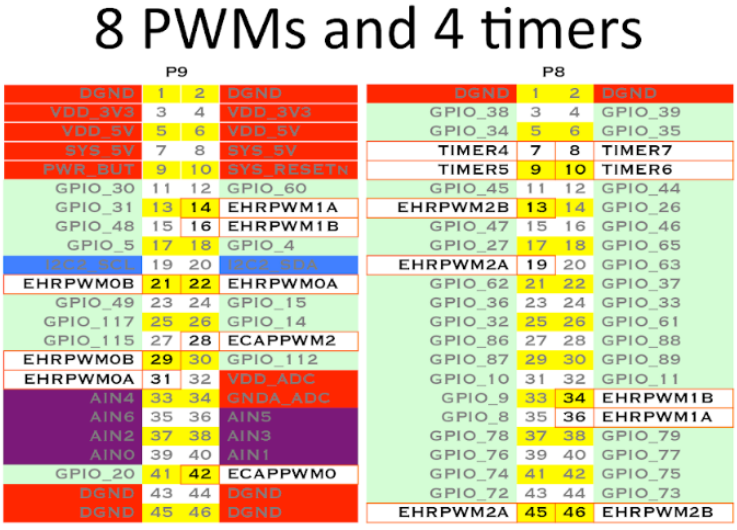

Figura 2.36. Pines PWM

Capítulo 3. Sistema de control de temperatura

A continuación, se procederá a implementar el sistema de comunicaciones desarrollado en un sistema de control de temperatura conocido como el "Process Control Trainer 37-100". El Process Control Trainer 37-100 es un equipo utilizado en el campo de la automatización industrial y está diseñado específicamente para el entrenamiento y la experimentación en el control de procesos. Este sistema combina elementos de hardware y software para simular un entorno industrial realista y permitir a los usuarios aprender y practicar los principios y técnicas de control de temperatura.

El sistema de control de temperatura del Process Control Trainer 37-100 se basa en la medición precisa de la temperatura utilizando sensores especializados y en la generación de una señal PWM (Modulación por Ancho de Pulso) para controlar un actuador, un calentador en este caso. La medición de temperatura se llevará a cabo en el interior de un tubo, dentro del cual circula un flujo de aire controlado por un ventilador situado en un extremo del tubo, y junto al cual está la resistencia calefactora. Con la PWM controlamos la intensidad de corriente que se aplica a esta resistencia. El sensor está situado en el extremo opuesto, por el cual sale dicho flujo de aire.

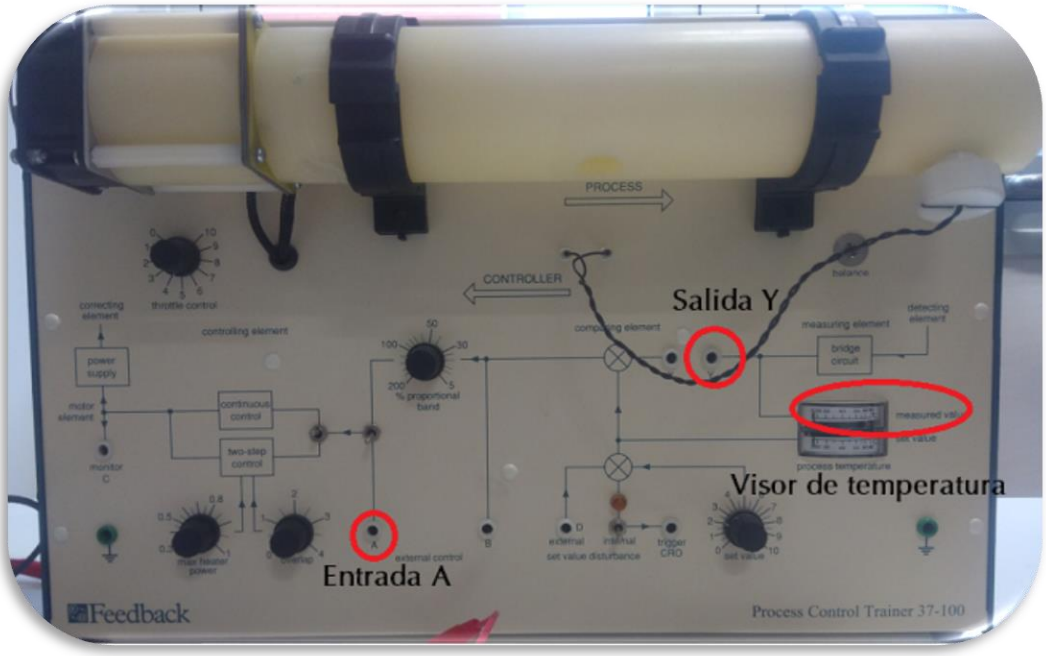

Figura 3.1. Process Control Trainer 37-100

En el sistema que se va a desarrollar, el control de la velocidad del ventilador se llevará a cabo de manera manual. En la imagen 3.1, se puede ver como el sistema dispone de múltiples funcionalidades y maneras de ejercer el control, pero para el sistema a implementar solo se va a usar la entrada A, que será la entrada del sistema. Aquí se ejercerá el control de la potencia aplicada a la resistencia mediante la señal PWM. Por otra parte, se va a usar la salida Y, que será la entrada analógica de la BBB. Aquí se medirá el voltaje de salida del sistema que se tenga en un instante concreto. Además, también se ve como se dispone de un visor de temperatura, del cual se hará uso para establecer una relación entre el voltaje de salida del sistema, en función de la temperatura.

El objetivo principal será desarrollar el control de la temperatura del sistema, utilizando la salida del sistema como retroalimentación y aplicando un control sobre la entrada. Para lograr esto, utilizaremos el lenguaje de programación C++ para crear una serie de clases que nos permitan implementar un controlador eficiente y preciso. Las variables clave que utilizaremos en nuestro controlador serán la lectura de temperatura del sistema, que obtenemos a través del sensor instalado, y la señal de control que aplicaremos al calentador. Estas variables serán la base para el diseño e implementación del algoritmo de control, que nos permitirá mantener la temperatura del sistema dentro de los rangos deseados. En el sistema, aunque el objetivo principal sea medir la temperatura, la salida que obtendremos del sistema será el voltaje proporcionado. Además, como la BeagleBone Black tiene una entrada analógica con un rango de voltaje de 0 a 1.8V necesitaremos convertir el voltaje de salida del sistema a este rango de valores. Para lograr esta conversión, sabiendo que el rango máximo de voltaje de entrada que puede suministrar la señal PWM es de 3.3V cuando el ciclo de trabajo es del 100%, hemos medido el voltaje de salida cuando se aplican 3.3V a la entrada del sistema. Para ajustar el voltaje de salida al rango de 0 a 1.8V, se implementará un divisor de tensión. Sabiendo que cuando se aplican 3.3V de entrada, se obtienen 7.8V de salida, hemos seleccionado resistencias de 7.3 Kohm y 2.2 Kohm para el divisor de tensión. Con esta configuración, obtendremos 1.8V a la salida del divisor de tensión, lo cual corresponderá a la entrada analógica de la BBB. De esta manera, mediante la implementación del divisor de tensión, lograremos adecuar el voltaje de salida del sistema al rango requerido por la entrada analógica de la BBB, permitiéndonos así obtener una medición precisa de la temperatura a partir del voltaje proporcionado.

$$Vout = \frac{R2}{R1 + R2} \cdot Vin$$

Figura 3.2. Partidor de tensión

A través del uso de las clases y métodos que desarrollaremos en C++, podremos realizar cálculos y manipulaciones necesarias para aplicar estrategias de control, como el control proporcional, integral y derivativo (PID), o cualquier otro algoritmo de control específico que consideremos adecuado para nuestro sistema. Además, estas clases nos brindarán la flexibilidad para ajustar y optimizar los parámetros de control, así como implementar funcionalidades adicionales.

3.1. Clase Control

La clase "Control" es una clase base que proporciona la estructura y funcionalidad para implementar un controlador en el proyecto. Esta clase tiene como objetivo principal ejercer el control sobre un sistema específico y calcular la acción de control necesaria para mantener la salida del sistema lo más cercana posible a una consigna deseada.

La clase "Control" cuenta con las siguientes variables miembro: "consigna" para almacenar el valor deseado de la salida, "salida" para almacenar la lectura actual de la salida del sistema, "error" para calcular la diferencia entre la consigna y la salida, y "accionControl" para almacenar la acción de control calculada.

Además, la clase "Control" proporciona métodos para obtener los valores de la consigna, la salida, el error y la acción de control. También se incluyen métodos para establecer la consigna y la salida. Estos métodos permiten configurar los valores de referencia y actualizar la lectura actual del sistema.

Cabe destacar que la clase "Control" es una clase abstracta, ya que declara un método virtual puro llamado "actualizar", el cual debe ser implementado por las clases derivadas. Este método se encargará de calcular la acción de control específica según el algoritmo de control utilizado, es decir, si el controlador es un PID, tendrá un algoritmo distinto al de un controlador todo o nada, por ejemplo. Al ser un método virtual puro, cada clase derivada debe proporcionar su propia implementación de este método.

```
#ifndef _CONTROL_
#define _CONTROL_

class Control {

protected:
  double consigna;
  double salida;
  double error;
  double accionControl;
public:
  Control();
  double getConsigna();
  double getSalida();
  double getError();
  double getAccionControl();
  double setConsigna(double c);
  double setSalida(double s);
  //Ejerce el control y devuelve la acción de control
  virtual double actualizar() = 0;
};
#endif
```

Figura 3.3. Clase Control.hpp

```
#include "Control.hpp"

Control::Control(){
}

// Obtener la consigna
double Control::getConsigna() {
  return this->consigna;
}

//Obtener la salida del sistema
double Control::getSalida() {
  return this->salida;
}

//Obtener el error
double Control::getError() {
  return this->error;
}

//Obtener la Accion de control
double Control::getAccionControl(){
  return this->accionControl;
}

//Establecer el valor de consigna
double Control::setConsigna(double c) {
    this->consigna = c;
    return this->consigna;
}

//Establecer el valor de salida
double Control::setSalida(double s) {
    this->salida = s;
    return this->salida;
}
```

Figura 3.4. Clase Control .cpp

3.2. Clase Control On-Off

La clase "ControlOnOff" es una clase derivada de la clase base "Control" y se encarga de implementar un controlador de tipo On-Off (encendido-apagado) en el proyecto. Este tipo de controlador toma decisiones binarias, es decir, solo tiene dos estados posibles: encendido o apagado.

La clase "ControlOnOff" hereda todas las variables miembro y métodos de la clase base "Control". Además, implementa su propio constructor llamado "ControlOnOff", el cual inicializa las variables miembros a valores predeterminados. En este caso, la consigna se establece en 0.5, puesto que será el valor de la entrada analógica que se desea obtener, es decir, un valor entre 0 y 1,8V. Además, el error se inicializa a 0, la acción de control se establece en 0 y la salida se establece en 0 también.

La clase "ControlOnOff" también implementa el método virtual puro "actualizar" heredado de la clase base. En este método, se calcula el error restando la salida actual de la consigna. Luego, se aplica una banda de histéresis para evitar la conmutación constante del controlador. Si el error es mayor a 0.2, se establece la acción de control en 0 (apagado). Si el error es menor a -0.2, se establece la acción de control en 1 (encendido). Finalmente, se devuelve el valor de la acción de control.

```
#ifndef _CONTROL_ON_OFF_
#define _CONTROL_ON_OFF_

#include "Control.hpp"

class ControlOnOff : public Control {

public:
  //Inicializa las variables a 0
  ControlOnOff();

  //Ejerce el control y devuelve las varibales a actualizar

  double actualizar();


};
#endif
```

Figura 3.5. Clase Control On-Off .hpp

```cpp
#include "ControlOnOff.hpp"

ControlOnOff::ControlOnOff() : Control(){
    this->consigna = 0.5;
    this->error = 0;
    this->accionControl = 0;
    this->salida = 0;
}

double ControlOnOff::actualizar() {
    this->error = this->salida - this->consigna;
    // Banda de histeresís para no conmutar todo el rato
    if( this->error > 0.2 ) {
        this->accionControl = 0; // Acción de control a 0
    }
    if( this->error < -0.2) {
        this->accionControl = 1; // Acción de control a 1
    }
    return accionControl;
}
```

Figura 3.6. Clase Control On-Off .cpp

```cpp
#include "ControlOnOff.hpp"
#include "BBB_PWM.hpp"
#include "BBB_AD.hpp"
#include <iostream>
#include <unistd.h> // para la función sleep()

int main() {
  // Crear objetos para las clases de PWM y AD
  BBB_PWM pwm("P9_14");
  BBB_AD a0(0);
  //Periodo de la PWM
  pwm.setPeriod(10000);
  pwm.setEnable(true);
  // Crear objeto de la clase ControlOnOff
  ControlOnOff control;

  // Establecer el valor de consigna
  control.setConsigna(1.4);

  while(true){
    // Leer la entrada analógica de 0V a 1,8V
    control.setSalida(a0.readV());
    //Mostrar valor
    std::cout << " Salida: " << control.getSalida() << std::endl;

    // Llamar al método actualizar() de la clase ControlOnOff
    double controlador = control.actualizar();

    // Ejercer la acción de control en la señal PWM
    pwm.setDutyPer1(controlador);

    sleep(1);
  }

  return 0;
}
```

Figura 3.7. Prueba Clase Control On-Off

Como se observa en la figura 3.7, se ha llevado a cabo una prueba para probar la clase Control On-Off así como la salida PWM, y este a sido el resultado, mostrando por pantalla el comando que se debe ejercer:

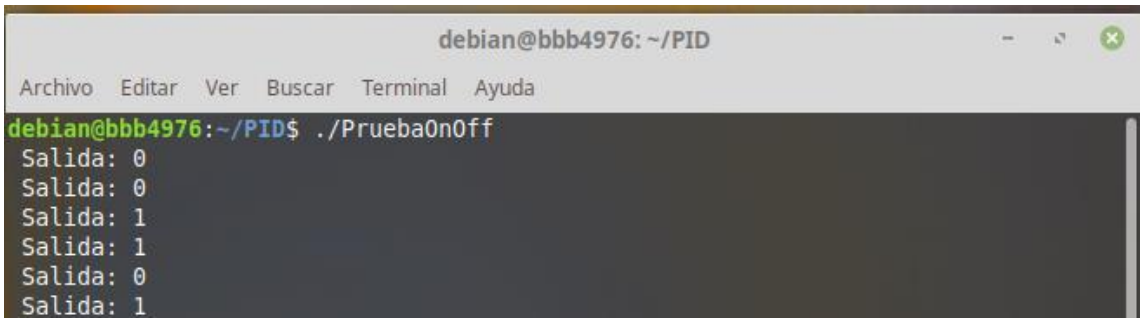

Figura 3.8. Ejecución de Prueba Clase Control On-Off

A pesar de que la clase ControlOnOff proporciona una solución básica para ejercer el control, no es la opción más adecuada para nuestro proyecto debido a su naturaleza de conmutación frecuente y su falta de precisión. Al utilizar un enfoque de encendido y apagado del actuador basado en una banda de histéresis, esta estrategia puede resultar en un comportamiento oscilante y fluctuaciones indeseables en el sistema.

En cambio, se requiere una opción más precisa y robusta que permita un control más suave y continuo sobre el actuador. Esto es especialmente importante cuando se trata de sistemas sensibles a las fluctuaciones y que requieren una respuesta más precisa, como el control de temperatura. Deseamos evitar cambios bruscos en la acción de control para minimizar las perturbaciones y mantener un rendimiento estable del sistema.

Para lograr este objetivo, exploraremos otras técnicas y algoritmos de control más sofisticados, como el control proporcional-integral-derivativo (PID). Esta alternativa nos permitirá ajustar y adaptar el control en función de la variación en la consigna y la respuesta del sistema, brindando una mayor precisión y estabilidad en el control de la temperatura.

3.3. Clase Control PID

A continuación, nos adentraremos en el desarrollo del algoritmo de control PID para nuestro sistema. Sin embargo, antes de comenzar, es fundamental comprender el tipo de sistema con el que estamos trabajando. Asumiremos que se trata de un sistema de primer orden, lo cual nos proporciona una base sólida para diseñar el controlador.

Para obtener una mejor comprensión del comportamiento del sistema, aplicaremos una entrada escalón al sistema y analizaremos su respuesta. Esta prueba nos permitirá obtener información valiosa sobre la dinámica del sistema, como su tiempo de respuesta, tiempo de establecimiento y posibles oscilaciones.

Mediante la aplicación de la entrada escalón y la observación de la respuesta del sistema, estaremos en condiciones de determinar los parámetros clave para nuestro controlador PID, como la constante de tiempo, la ganancia, y el retardo del sistema. Estos parámetros son fundamentales para ajustar adecuadamente el controlador y lograr un rendimiento óptimo del sistema de control de temperatura.

Una vez que hayamos realizado esta caracterización preliminar del sistema, estaremos preparados para aplicar el algoritmo de control PID. Este algoritmo combina tres componentes principales: proporcional, integral y derivativo, para proporcionar un control más preciso y robusto. Ajustaremos cuidadosamente los parámetros del controlador PID en base a la respuesta del sistema a la entrada escalón, con el objetivo de minimizar el error, reducir el tiempo de respuesta y mantener una respuesta estable y suave del sistema ante cambios en la consigna.

3.3.1 Prueba para entrada escalón

En el siguiente paso, llevaremos a cabo una prueba específica para obtener los datos de la salida del sistema al aplicar una señal escalón a la entrada. En este caso, utilizaremos la señal PWM con un *duty cycle* del 100%. El propósito de esta prueba es recopilar información relevante sobre la respuesta del sistema a la entrada escalón y utilizarla en etapas posteriores del proyecto.

Durante la prueba, registraremos los valores de salida del sistema en un archivo para su posterior análisis.

```
int main() {

    BBB_AD a0(0);
    BBB_PWM pinPWM("P9_14");

    muestraPin(pinPWM);

    pinPWM.setPeriod(10000);
    pinPWM.setEnable(true);
    muestraPin(pinPWM);

    // Establecer la PWM al %
    double per1;
    std::cout << "Elija el valor para la PWM en tanto por 1 " << std::endl;
    std::cin >> per1;
    std::cin >> fichero;

    // Guardar el tiempo actual
    auto start = std::chrono::high_resolution_clock::now();

    // Guardar el valor de a0.readV() junto con el tiempo en el archivo de datos
    std::ofstream outfile(fichero, std::ios_base::app);

    // Bucle while para recopilar los datos y escribirlos en el archivo
    while (true) {
        // Calcular el tiempo transcurrido
        auto now = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = now - start;

        // Salir del bucle después de 900 segundos (20 min)
        if (elapsed.count() >= 1200)
            break;

        // Guardar los valores del tiempo y el valor analógico en el flujo de fichero
        outfile << elapsed.count() << "," << a0.readV() << std::endl;

        // Pausa de 1 milisegundo
        std::this_thread::sleep_for(std::chrono::seconds(1));

        // Establecer la PWM después de 60 segundos
        if (elapsed.count() >= 60) {
            pinPWM.setDutyPer1(per1);
            muestraPin(pinPWM);
        }
    }

    outfile.close();

    return 0;
}
```

Figura 3.9. Prueba para entrada escalón

Una vez que dispongamos del fichero de datos con los valores del voltaje de salida del sistema en función del tiempo, procederemos a realizar su representación gráfica utilizando Octave.

Utilizaremos las capacidades gráficas de Octave para crear una visualización clara y comprensible de los datos obtenidos durante la prueba. A través de la representación gráfica, podremos observar la evolución del voltaje de salida a lo largo del tiempo y analizar su comportamiento en relación con la señal escalón aplicada.

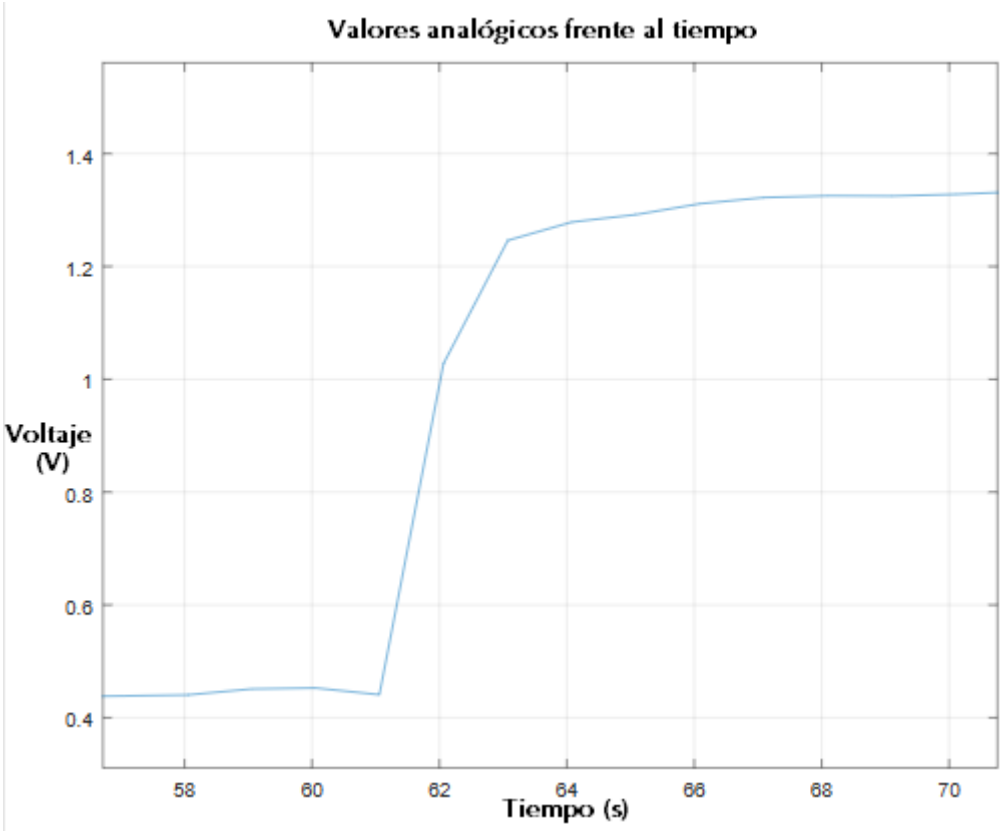

Figura 3.10. Respuesta del sistema a la entrada escalón

Como se puede observar en la figura generada, el comportamiento del sistema durante el transitorio exhibe una respuesta que se asemeja a una función exponencial. Esta característica es fundamental para el cálculo de los parámetros del sistema.

A partir de la curva obtenida en la representación gráfica, podremos realizar análisis y cálculos específicos para determinar los parámetros del sistema de primer orden. Estos parámetros incluyen la constante de tiempo (τ) y la ganancia del sistema (K).

La constante de tiempo representa la velocidad con la que el sistema alcanza aproximadamente el 63.2% de su valor final después de aplicar una señal escalón. Mientras tanto, la ganancia del sistema es la relación entre la magnitud de la respuesta del sistema y la magnitud de la señal de entrada.

Al calcular y obtener estos parámetros, podremos tener una mejor comprensión del comportamiento y las características del sistema. Esto nos permitirá diseñar un controlador PID más preciso y ajustado, que tome en cuenta las peculiaridades y dinámicas específicas del sistema de control de temperatura.

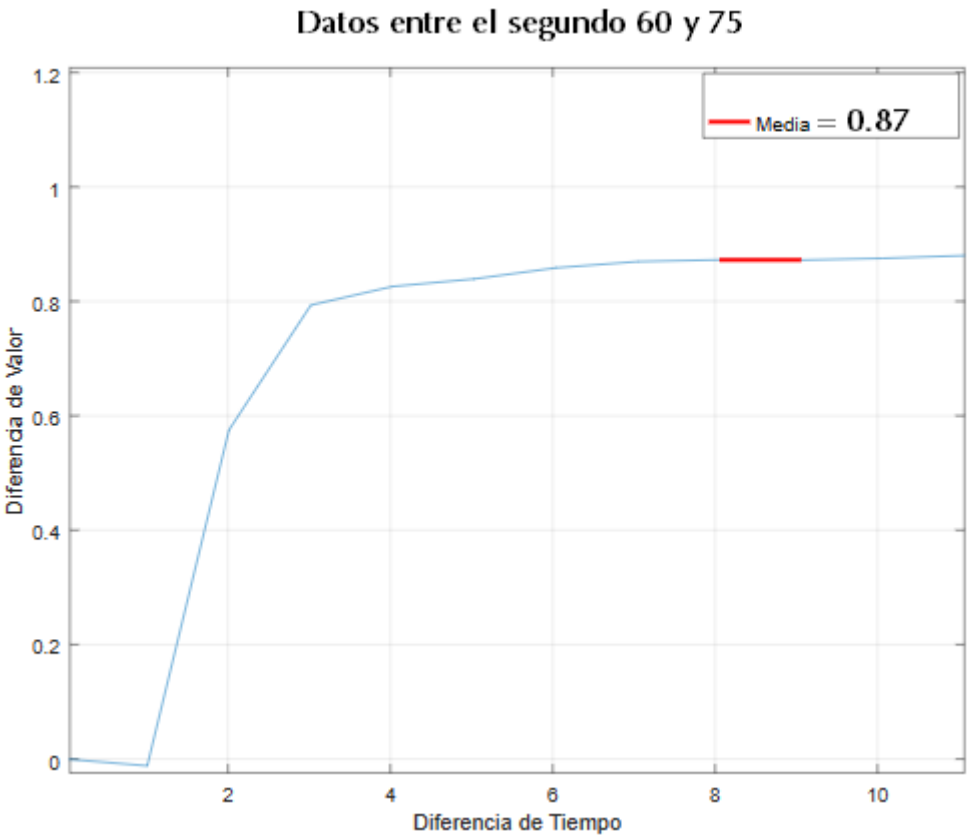

Figura 3.11. Respuesta del sistema en diferencias

En este caso particular, se ha configurado la señal PWM al 0% y se ha aplicado el escalón de un 100% en el segundo 60. Esta elección nos permite obtener datos tanto del estado previo al escalón como de la respuesta del sistema después de aplicarlo. Además, podemos observar en la figura 3.11 que la respuesta del sistema no se produce justo cuando se le aplica el escalón, sino que tarda aproximadamente 1 segundo en responder, es decir, este será el retardo del sistema.

Después del escalón, al establecer la señal PWM al máximo, se espera que el sistema alcance un valor constante en su salida.

Al analizar la respuesta del sistema al escalón, se ha considerado el instante en el que se produce el salto como el tiempo t=0. Esto nos permite tener una referencia clara para el análisis y cálculo de los parámetros del sistema.

Además, se ha tomado la media del valor estacionario antes del escalón y se ha considerado como el punto de referencia 0 para los valores de la respuesta del sistema. Esto nos permite tener una base relativa desde la cual medir la variación y el comportamiento del sistema a partir del escalón aplicado.

Con Octave, aplicaremos el criterio del 63% y el 28% del valor final para determinar la constante de tiempo (τ) del sistema. Es decir, se calculan los tiempos en el cual la respuesta del sistema alcanza el 63% de su valor final, es decir, el 63% de 0.87, así como el tiempo donde alcanza el 28%, y con estos tiempos seremos capaces de calcular τ, así como el tiempo muerto (tm).

$$\tau = \frac{3}{2}(t^{63} - t^{28}) \qquad (1.1)$$

$$tm = t^{63} - \tau \qquad (1.2)$$

Al realizar los cálculos, hemos determinado que el sistema alcanza el 63% de su valor final a los 1,96 segundos y el 28% a los 1,44 segundos. Sustituyendo estos tiempos en las fórmulas correspondientes, podemos calcular la constante de tiempo y el tiempo muerto del sistema. En este caso, hemos obtenido una constante de tiempo de 0,78 segundos y un tiempo muerto de 1,18 segundos.

La constante de tiempo representa la velocidad a la cual el sistema se acerca a su valor final, mientras que el tiempo muerto representa el tiempo que tarda el sistema en comenzar a responder después de que se ha aplicado una entrada.

Estos valores de la constante de tiempo y el tiempo muerto son orientativos para el diseño del controlador PID, ya que nos proporcionan información sobre la respuesta dinámica del sistema.

Por otra parte, la ganancia del sistema (k), es la relación entre la entrada y la salida del sistema. En nuestro caso, la entrada se ha aplicado como una señal escalón utilizando la PWM, variando desde el 0% hasta el 1% de duty cycle. Por

otro lado, la salida del sistema se ha estabilizado en un valor de 0,87 en el estado estacionario. Lo que significa que la ganancia es directamente 0,87.

Una vez que hemos determinado la ganancia, el tiempo muerto y la constante de tiempo de nuestro sistema de primer orden, disponemos de información importante para calcular los parámetros del controlador PID.

3.3.2 Algoritmo del controlador PID

La clase "ControlPID" hereda de la clase base Control y se encarga de implementar el controlador PID para el sistema de control de temperatura. Esta clase cuenta con varias variables y funciones que son necesarias para el cálculo y ajuste de los parámetros del controlador.

En cuanto a las variables, se encuentran los coeficientes de ganancia Kp, Ki y Kd, que representan la ganancia proporcional, integral y derivativa respectivamente. Además, se tienen variables para el almacenamiento de errores anteriores, el valor de integración y el tiempo de muestreo.

El constructor de la clase inicializa todas las variables a cero, incluyendo las heredadas de la clase base. También se establece un valor predeterminado para el tiempo de muestreo.

La función "menuSeleccion" permite al usuario modificar los parámetros del controlador PID, como Kp, Ki, Kd, el tiempo de muestreo y la consigna. Se muestra un menú de opciones y se solicita al usuario ingresar el valor deseado para cada parámetro.

La función "actualizar" se encarga de calcular la acción de control del controlador PID. Se calcula el error actual, la acción proporcional, la acción integral y la acción derivativa. Luego, se suman estas tres partes para obtener la acción de control final. Además, se realiza un límite para asegurar que la acción de control esté dentro del rango permitido (0 a 1). Por último, se actualizan los errores anteriores.

La clase también proporciona funciones para obtener y establecer los valores de Kp, Ki, Kd y el tiempo de muestreo.

```hpp
#ifndef _PID_
#define _PID_

#include "Control.hpp"


class ControlPIDnuevo : public Control {
private:
  double Kp;
  double Ki;
  double Kd;
  double error_previo;
  double error_anterior;
  double integralError;
  double Ts; // Tiempo de muestreo en segundos
public:
  //Inicializa las variables a 0
  ControlPIDnuevo();
  //Menu para cambiar los parámetros
  void menuSeleccion();
  double getKp();
  double getKi();
  double getKd();
  double getTmuestreo();
  double setKp(double Kp);
  double setKi(double Ki);
  double setKd(double Kd);
  double setTmuestreo(double Ts);
  double actualizar();
};

#endif
```

Figura 3.12. ControlPID .hpp

```cpp
#include "ControlPIDnuevo.hpp"
#include <iostream>

ControlPIDnuevo::ControlPIDnuevo() : Control(){
  this->consigna = 0;
  this->error = 0;
  this->accionControl = 0;
  this->salida = 0;
  this->Kp = 0;
  this->Ki = 0;
  this->Kd = 0;
  this->error_previo = 0;
  this->error_anterior = 0;
  this->Ts = 30000; //Tiempo de muestreo en microsegundos
  this->integralError = 0;
}

void ControlPIDnuevo::menuSeleccion(){
  double nuevo_Kp, nuevo_Ki, nuevo_Kd, nuevo_Ts, nueva_Consigna;
  char input;

  // Mostramos opciones y esperamos input por teclado
  std::cout << "Seleccione valor a modificar: 1-Kp, 2-Ki, 3-Kd, 4-Ts, 5-Consigna" << std::endl;
  std::cin >> input;

  switch(input) {
    case '1':
      std::cout << "Introduzca nuevo valor para Kp:" << std::endl;
      std::cin >> nuevo_Kp;
      this->Kp = nuevo_Kp;
      break;
    case '2':
      std::cout << "Introduzca nuevo valor para Ki:" << std::endl;
      std::cin >> nuevo_Ki;
      this->Ki = nuevo_Ki;
      break;
```

```
    case '3':
      std::cout << "Introduzca nuevo valor para Kd:" << std::endl;
      std::cin >> nuevo_Kd;
      this->Kd = nuevo_Kd;
      break;
    case '4':
      std::cout << "Introduzca nuevo valor para Ts:" << std::endl;
      std::cin >> nuevo_Ts;
      this->Ts = nuevo_Ts;
      break;
     case '5':
      std::cout << "Introduzca nuevo valor para consigna:" << std::endl;
      std::cin >> nueva_Consigna;
      this->consigna = nueva_Consigna;
      break;
    default:
      std::cout << "Opción inválida, no se realizará ninguna acción." << std::endl;
      break;
  }

}


double ControlPIDnuevo::actualizar(){

  this->error = this->consigna - this->salida;

  // Acción proporcional
  double p = this->Kp * this->error;

  // Integral del error
  this->integralError += this->error;

  // Acción integral
  double i = this->Ki * this->integralError;
  // Acción derivativa
  float derivada = this->Kd * ( (this->error - this->error_previo ) / this-> Ts );

  // Sumamos las tres partes
  this->accionControl = p + i + derivada;

  if ( accionControl < 0 ){
      accionControl = 0;
  }
  else if (accionControl > 1){
      accionControl = 1;
  }


  // Actualizamos los errores anteriores
  this->error_previo = this->error;

  return this->accionControl;
}

//Obtener la Kp
double ControlPIDnuevo::getKp(){
  return this->Kp;
}

//Establecer el valor de Kp
double ControlPIDnuevo::setKp(double Kp) {
    this->Kp = Kp;
    return this->Kp;
}
```

Figura 3.13. ControlPID .cpp

En la figura 3.13, se han omitido las funciones para obtener y establecer los valores de las variables, puesto que son iguales a las funciones para la Kp que se muestran en la imagen.

Los parámetros del controlador PID implementados en la clase ControlPIDnuevo han sido obtenidos utilizando el método de Ziegler-Nichols, que se muestra en la figura 3.14. En nuestro caso, una vez que conocemos los parámetros del sistema de temperatura, como la constante de tiempo, el tiempo muerto y la ganancia, aplicamos el método de Ziegler-Nichols para obtener los parámetros del controlador PID.

Tipo de Controlador	K_p	T_i	T_d
P	$\frac{1}{K}\frac{T}{t_m}$		
PI	$\frac{0.9}{K}\frac{T}{t_m}$	$3,33t_m$	
PID	$\frac{1.2}{K}\frac{T}{t_m}$	$2t_m$	$0,5t_m$

Figura 3.14. Parámetros del controlador con Ziegler-Nichols

Hemos obtenido unos valores aproximados para los parámetros del controlador PID en nuestro sistema de temperatura. En base a Ziegler-Nichols, hemos establecido los siguientes valores: Kp = 0.9, Ki = 2.4 y Kd = 0.6. Sin embargo, es importante tener en cuenta que estos valores son orientativos y que el sistema tiene sus propios requisitos y características. Por lo tanto, aunque hemos proporcionado valores iniciales para los parámetros del controlador PID, es probable que sea necesario realizar ajustes y afinamientos adicionales para adaptarlos específicamente a nuestro sistema y lograr el control deseado de la temperatura.

Se ha llevado a cabo una prueba para evaluar el rendimiento del controlador PID en función de los valores analógicos de salida del sistema, es decir, la entrada analógica. Durante la prueba, se mostraron en pantalla los valores del controlador PID, que corresponden al ancho del pulso de la señal PWM.

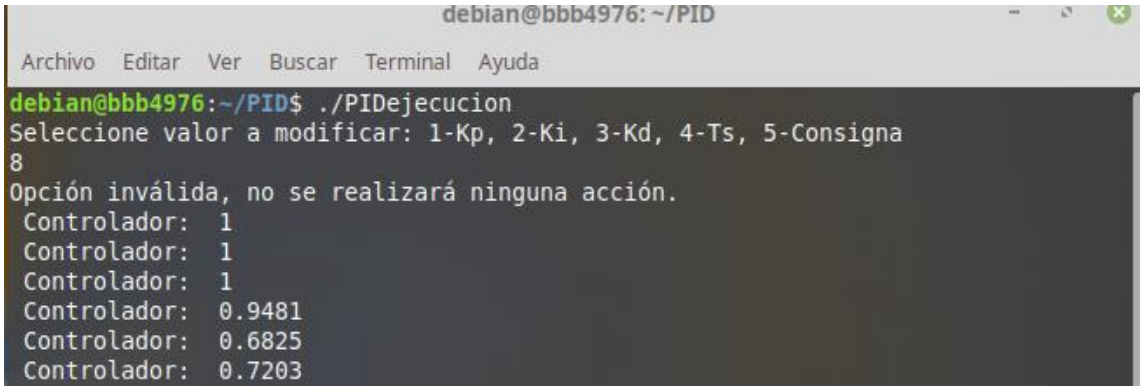

Figura 3.15. Control PID comando PWM

Como se puede observar en la imagen resultante, los valores del controlador PID se encuentran en el rango del 0% al 1% de duty cycle. Estos valores representan la señal de control proporcionada al sistema para mantener el valor de voltaje deseado, es decir, la temperatura deseada.

En el proceso de desarrollo del controlador, es importante tener en cuenta que no es necesario conocer directamente la temperatura real del sistema en cada momento. En lugar de eso, nos enfocamos en conocer el voltaje de salida.

Cuando queramos obtener una temperatura deseada, es decir, la consigna del sistema, necesitaremos conocer la relación entre el voltaje de salida y la temperatura. Esto implica realizar previamente una calibración del sistema, donde se establece una correspondencia entre los valores de voltaje medidos y las temperaturas correspondientes, que veremos más adelante. Esta relación nos permite establecer la consigna deseada en términos de voltaje de salida.

Una vez que conocemos esta relación, el controlador PID trabaja con la consigna de voltaje establecida y ajusta la señal de control para alcanzar y mantener dicha consigna, de esta manera, el controlador se encarga de regular la temperatura del sistema.

Esta sería la jerarquía entre las clases que aplican el control:

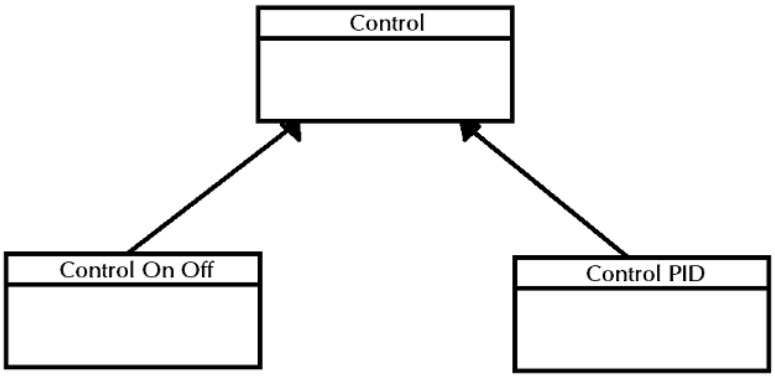

Figura 3.16. Diagrama UML clases de control

3.5. Desarrollo de las comunicaciones

La implementación propuesta para el sistema de control de temperatura es la siguiente:

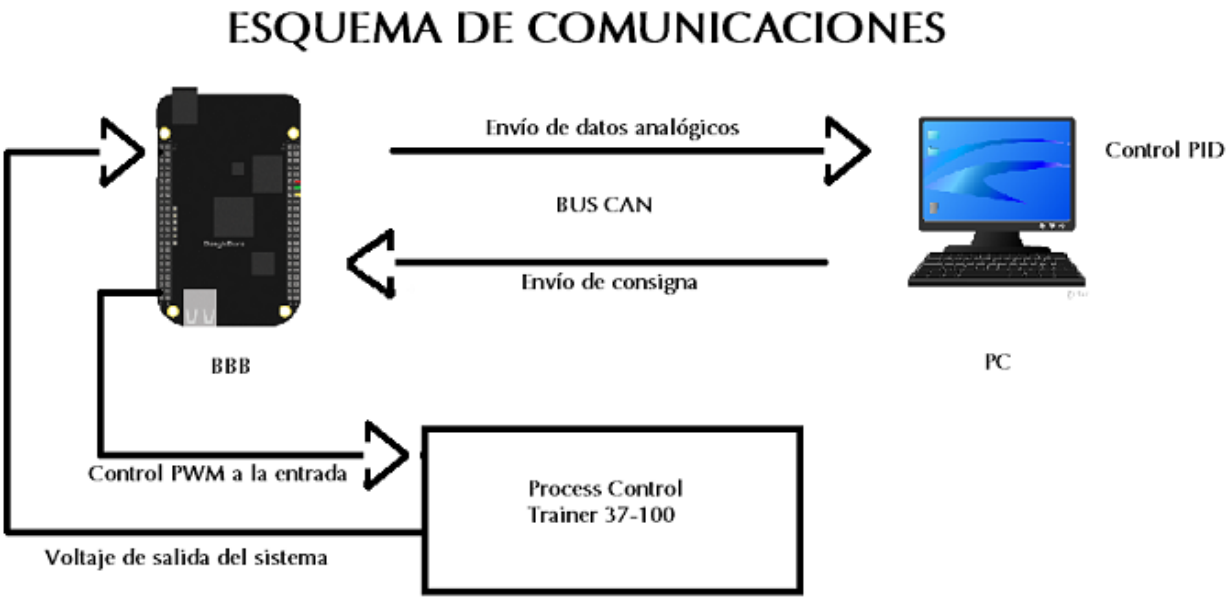

Figura 3.17. Esquema de comunicaciones

La lectura de los datos de salida del sistema, es decir, el voltaje de salida, se realizará desde la BeagleBone Black. Estos datos serán transmitidos a través del bus CAN hacia el ordenador. En el ordenador, se llevará a cabo el cálculo del controlador PID correspondiente utilizando los datos recibidos. Una vez obtenida la señal de control, esta será enviada de regreso a la BBB a través del bus CAN. Los paquetes con el comando de la señal PWM, que representa la entrada del sistema, serán capturados por la BBB para ejercer el control deseado sobre el sistema.

Esta comunicación bidireccional a través del bus CAN permite establecer una conexión efectiva entre el ordenador y la BBB, lo que facilita el procesamiento y ajuste del controlador en el ordenador, mientras que la BBB se encarga de ejecutar el control en tiempo real sobre el sistema de temperatura.

3.4.1 Envío de datos analógicos

```
#include <iostream>
#include <iomanip> // para usar std::hex y std::setw
#include <sstream>
#include <string>
#include <unistd.h> // para usar la función sleep()
#include "Frame.hpp"
#include "BBB_AD.hpp"
#include "CAN.hpp"

int main() {

//Crear objeto para ADC
    BBB_AD a0(0);

//Crear objeto para elegir controlador
    CAN can(0);

//Crear objeto para enviar trama
    Frame trama(can.getSocket());

//Bucle para enviar tramas cada segundo
    while(true){

    // Obtener el valor raw del ADC
        int valorRaw = a0.read();

    // Obtener los valores de readPer1 y multiplicarlos por 1.8V para pasarlo
    // al rango de 0 a 1,8, y multiplicarlo por 100 para no tener decimales
        int valorPer1 = a0.readPer1() * 100 * 1.8;

    // Crear la cadena de caracteres que representa los datos en hexadecimal
        std::stringstream datosHex;
        datosHex << std::setfill('0') << std::setw(4) << std::hex << valorRaw
                 << std::setw(2) << std::hex << valorPer1;

        std::string datos = datosHex.str();
    //  Enviar trama
        trama.createFrame(0x100, datos);
        trama.enviarTrama();

    //  Mostrar los datos analógicos leidos
        std::cout << "Lectura Analógica raw: " << a0.read() << std::endl;
        std::cout << "Lectura Analógica %1: " << a0.readPer1() << std::endl;
        std::cout << "Lectura Analógica V: " << a0.readV() << std::endl;

    //  Esperar un segundo antes de enviar la siguiente trama
        sleep(1);
      }

  return 0;
}
```

Figura 3.18 Prueba envío datos analógicos

En primer lugar, se realizará una prueba para enviar los datos analógicos a través del bus CAN. Para ello, se utilizará la clase mencionada en la figura 3.18.

El objetivo es leer el valor analógico utilizando el ADC de la BeagleBone Black y transmitir ese valor a través del bus CAN.

En la prueba, se creará un objeto de la clase BBB_AD, que representa el ADC de la BBB, y se utilizará para leer el valor analógico. A continuación, se creará un objeto de la clase CAN para establecer la comunicación a través del bus CAN. Posteriormente, se creará un objeto de la clase Frame para crear una trama CAN que contendrá el valor analógico leído.

Dentro de un bucle, se realizará la lectura del valor analógico y se formateará adecuadamente para su transmisión. La trama CAN se creará con un identificador específico y los datos analógicos en formato hexadecimal. Luego, la trama se enviará a través del bus CAN utilizando el objeto de la clase Frame.

Durante la prueba, se mostrarán por pantalla los valores analógicos leídos para verificar su correcta lectura. Esto permitirá comprobar que los datos se están transmitiendo correctamente a través del bus CAN.

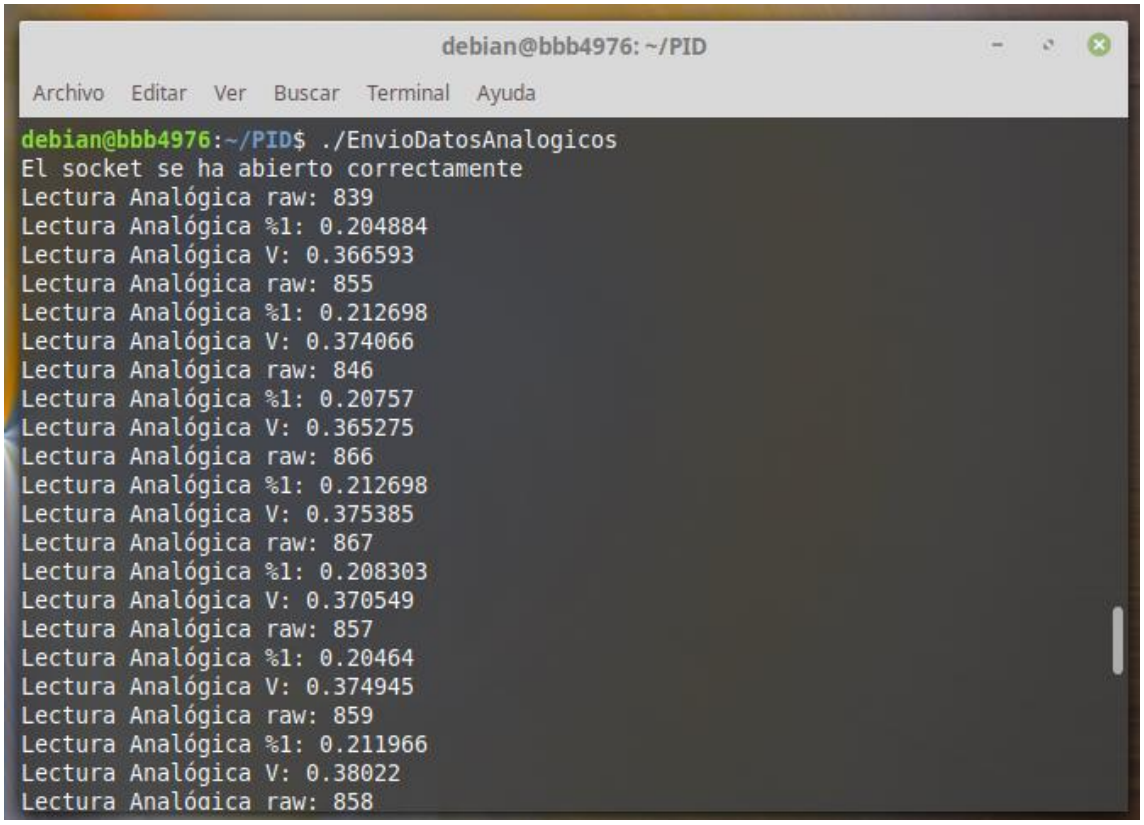

Figura 3.19 Ejecución prueba envío datos analógicos en BBB

En la figura 3.19 podemos observar como se muestran por pantalla los valores analógicos leídos. En la figura 3.20 hemos usado el comando “candump” en el ordenador para recibir las tramas con los valores.

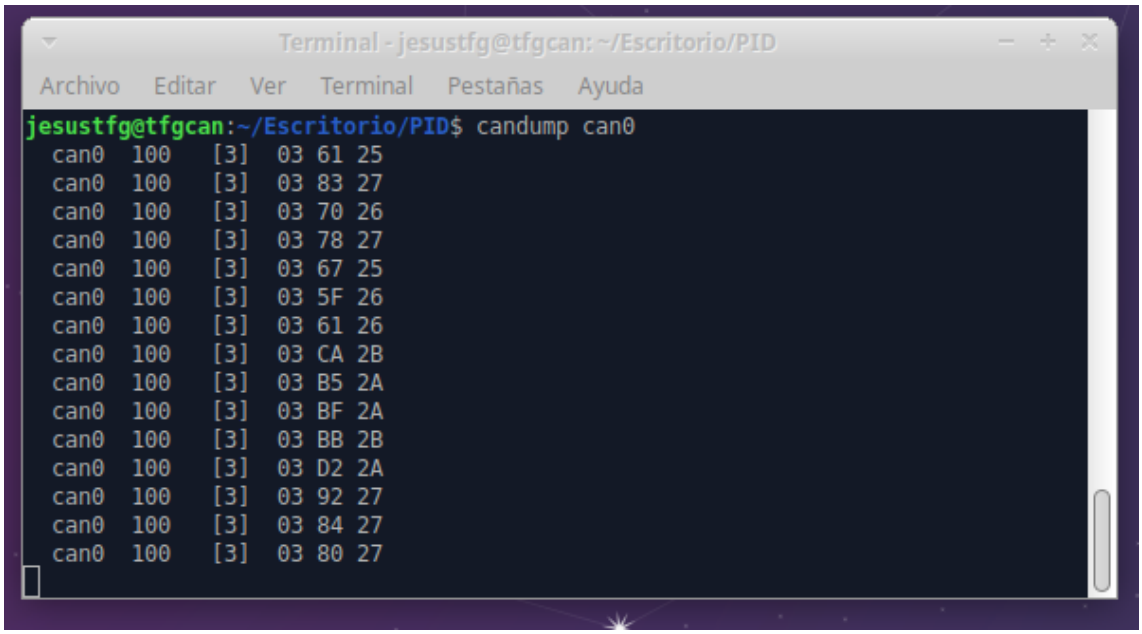

Figura 3.20 Recepción datos analógico en PC

Vemos como los dos últimos dígitos transmitidos en la trama son el valor hexadecimal del valor analógico, es decir, por ejemplo, la primera trama 25 en hexadecimal, que es 37 en decimal.

3.4.2 Recepción de datos analógicos

Una vez se han conseguido enviar los datos analógicos por el bus, lo siguiente que se realizará es una prueba para recibir las tramas de datos a través del bus CAN en el PC. En el código proporcionado, se utiliza la clase CAN para establecer la comunicación y la clase Frame para recibir las tramas CAN.

En el programa principal, se crea un objeto de la clase CAN para establecer la comunicación en el bus CAN. Luego, se crea un objeto de la clase Frame para recibir y procesar las tramas recibidas.

Dentro de un bucle infinito, se llama al método “recibirGuardarTrama” de la clase Frame para recibir una trama del bus CAN y almacenarla en el objeto. Este método ha sido implementado como novedad en la clase Frame.

```
void Frame::recibirGuardarTrama() {
    ssize_t nbytes = read(this->socket, &frame, sizeof(struct can_frame));

    if (nbytes < 0 || static_cast<unsigned>(nbytes) < sizeof(struct can_frame))
        std::cerr << "La trama está vacía o incompleta" << std::endl;
        exit(0);
    }
}
```

Figura 3.21 Método recibirGuardarTrama clase Frame

A continuación, se obtienen los datos de la trama en formato hexadecimal utilizando el método getdata, de la clase Frame. En este caso, se extrae la parte de la trama correspondiente al valor “valorPer1”, que se encuentra en los primeros 2 bytes de la trama. Luego, se convierte ese valor hexadecimal a decimal.

```
#include <iostream>
#include <iomanip>
#include <string>
#include "Frame.hpp"
#include "CAN.hpp"

int main() {
    // Crear objeto para elegir controlador
    CAN can(0);

    // Crear objeto para recibir trama
    Frame trama(can.getSocket());

    // Bucle para recibir tramas continuamente
    while (true) {
        // Recibir trama
        trama.recibirGuardarTrama();

        // Obtener los datos de la trama en formato hexadecimal
        std::string tramaHex = trama.getdata();

        // Extraer la parte de la trama correspondiente a valorPer1
        std::string valorPer1Hex = tramaHex.substr(4, 2);

        // Convertir el valorPer1 de hexadecimal a decimal
        int valorPer1Int = std::stoi(valorPer1Hex, nullptr, 16);

        // Calcular el valorPer1 en double dividiendo por 100
        double valorPer1Double = static_cast<double>(valorPer1Int) / 100.0;

        // Mostrar el valorPer1
        std::cout << "ValorPer1: " << valorPer1Double << std::endl;
    }

    return 0;
}
```

Figura 3.22 Prueba de recepción de datos analógicos

Después, se calcula el valor en formato double, y dividiendo por 100.0, para obtener el valor analógico en el rango de 0 a 1,8V. Por último, se muestra por pantalla el valor obtenido a partir de la trama recibida.

Lo siguiente que se ha hecho es ejecutar este código en el PC, y el código de envío de datos desde la BBB, obteniendo los datos transmitidos por pantalla, como se muestra en la figura 3.23.

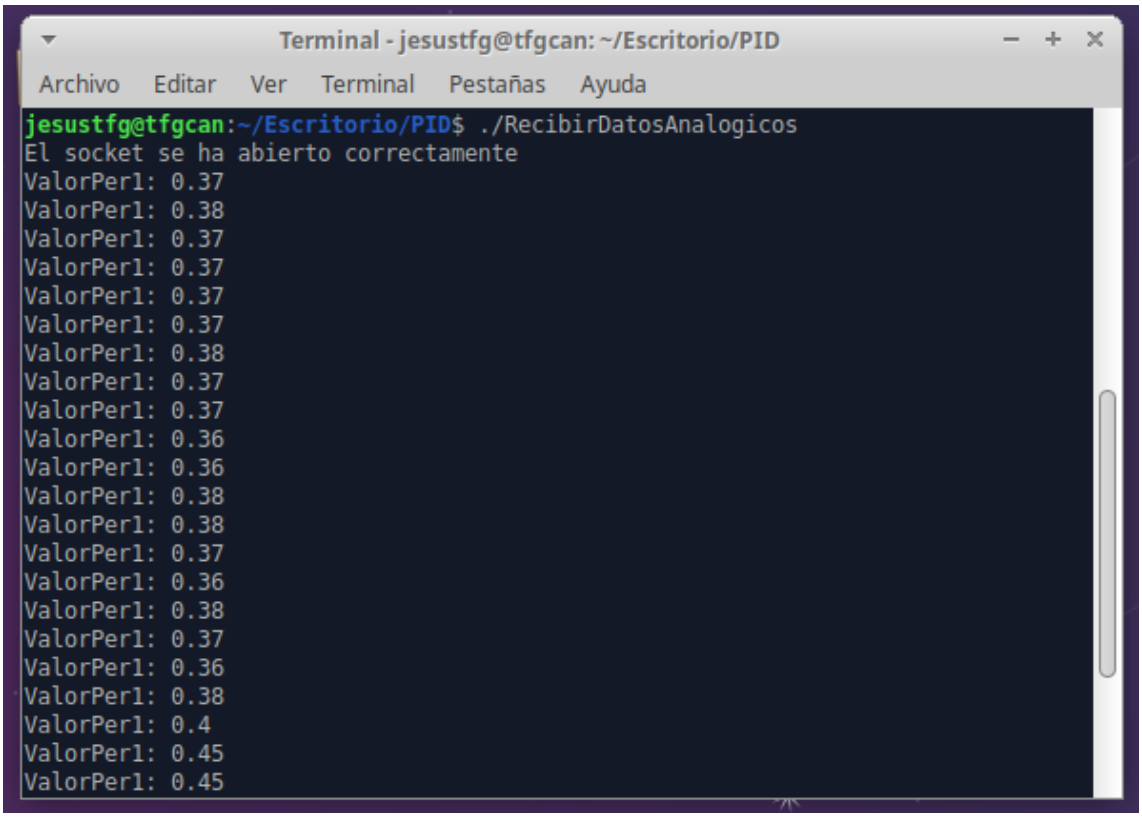

Figura 3.23. Ejecución prueba de recepción de datos analógicos en PC

3.4.3 Prueba Control PID

Una vez hemos transmitido los datos analógicos, vemos que se podría implementar de una manera parecida la forma de enviar la consigna generada por el código del controlador PID. Por lo tanto, se ha realizado la prueba de control PID en el ordenador, y se ha incluido la parte de recepción de recepción de datos analógicos, así como se ha añadido el envío de la consigna por el bus.

```cpp
#include "ControlPIDnuevo.hpp"
#include <iostream>
#include <thread>
#include "Frame.hpp"
#include "CAN.hpp"
#include <iomanip>


// Crear objeto para la clase ControlPID
ControlPIDnuevo PID;

//Crear objeto para el controlador CAN 0
CAN can(0);
//Crear objeto para el controlador CAN 1
CAN can1(1);

//Crear objeto para la clase Frame para recibir el dato analogico
Frame tramaDatAnalog(can.getSocket());

//Crear objeto para la clase Frame para enviar la consigna
Frame tramaConsigna(can1.getSocket());


void cicloControl(){

    //Definir el filtro para recibir tramas en el controlador can0
    struct can_filter rfilter [1];
    rfilter[0].can_id = 0x100;
    rfilter[0].can_mask = CAN_SFF_MASK;

    setsockopt(can.getSocket(), SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
  while(true){
      // Recibir trama
    tramaDatAnalog.recibirGuardarTrama();

    // Obtener los datos de la trama en formato hexadecimal
    std::string tramaHex = tramaDatAnalog.getdata();

    // Extraer la parte de la trama correspondiente a valorPer1
    std::string valorPer1Hex = tramaHex.substr(4, 2);

    // Convertir el valorPer1 de hexadecimal a decimal
    int valorPer1Int = std::stoi(valorPer1Hex, nullptr, 16);

    // Calcular el valorPer1 en double dividiendo por 100
    double valorPer1Double = static_cast<double>(valorPer1Int) / 100.0;

    // Mostrar el valorPer1 antes de asignarlo a setSalida
    std::cout << "ValorPer1 recibido: " << valorPer1Double << std::endl;

    // Leer la entrada analógica de 0V a 1,8V
    PID.setSalida(valorPer1Double);

    // Llamar al método actualizar() de la clase ControlPID
    double comando = PID.actualizar();
    //Convertir el comando a hexadecimal para enviarlo
    std::stringstream valorHex;
    valorHex << std::setfill('0') << std::to_string(comando) << std::setw(2)
             << std::hex;
    std::string valorString = valorHex.str();
    // Enviar Consigna a la BBB
    tramaConsigna.createFrame(253, valorString);
    tramaConsigna.enviarTrama();
    // Esperar un momento antes de la siguiente lectura
    std::this_thread::sleep_for(std::chrono::microseconds(static_cast<int>(PID.getTmuestreo())));
  }

}


int main() {

  // Establecer el valor de consigna
  PID.setConsigna(0.6);
  // Establecer el valor de Kp
  PID.setKp(0.8967);
    // Establecer el valor de Ki
  PID.setKi(2.4076);
    // Establecer el valor de Kd
  PID.setKd(0.6019);
  // Llamar al método actualizar() de la clase ControlPID en un hilo (t) separado
  std::thread t(cicloControl);

  while (true) {

    // Ejecutar la función menuselección()
    PID.menuSeleccion();

    // Mostrar por pantalla los datos del controlador
    std::cout << "Salida: " << PID.getSalida() << " Controlador: "
              << PID.getAccionControl() << " Error: " << PID.getError()
              << " Consigna: " << PID.getConsigna() << std::endl;
```

Figura 3.24 Prueba Control PID en PC

Primero se crean los objetos necesarios, como PID de la clase ControlPIDnuevo, “can” y “can1” de la clase CAN, y “tramaDatAnalog” y “tramaConsigna” de la clase Frame. Estos objetos se utilizarán para el control PID, la comunicación CAN y el envío y recepción de tramas. Después se define la función “cicloControl”, que se ejecutará en un hilo de ejecución (thread) separado que se encargue de ejecutar el ciclo de control con la frecuencia de muestreo establecida, permitiendo que el hilo principal interactúe con la persona usuario y le permita cambiar los parámetros del controlador. En este ciclo, se realiza lo siguiente:

- Se define el filtro para obtener solo tramas con los datos analógicos en el controlador can0.
- Se recibe una trama del bus CAN y se extraen los datos relevantes, en este caso el valor valorPer1.
- Se convierte valorPer1 de hexadecimal a decimal y se calcula su valor en formato double.
- Se establece este valor como entrada analógica para el controlador PID utilizando el método “setSalida” que es el método que hereda de la clase “Control”.
- Se llama al método actualizar de la clase ControlPIDnuevo para obtener el comando de control.
- El comando se convierte a formato hexadecimal y se envía como consigna a través del bus CAN utilizando el objeto tramaConsigna.

En la función principal main, se establecen los valores de la consigna y los parámetros del controlador PID utilizando los métodos correspondientes de la clase ControlPIDnuevo.

Luego se crea un hilo “t” que ejecuta la función cicloControl. Dentro del bucle principal, se muestra por pantalla la información relevante del controlador PID, como la salida, la acción de control, el error y la consigna. También se muestra por pantalla los parámetros del controlador PID, como los valores de Kp, Kd, Ki y el tiempo de muestreo (Ts). Finalmente, el programa se ejecutará indefinidamente en el bucle principal. En este caso hemos tomado un tiempo de muestreo aproximado a $\tau/20$.

En líneas abiertas, para evitar el hilo se podrían enviar los nuevos parámetros que queremos aplicar al control también por el bus, y así tener mas flexibilidad.

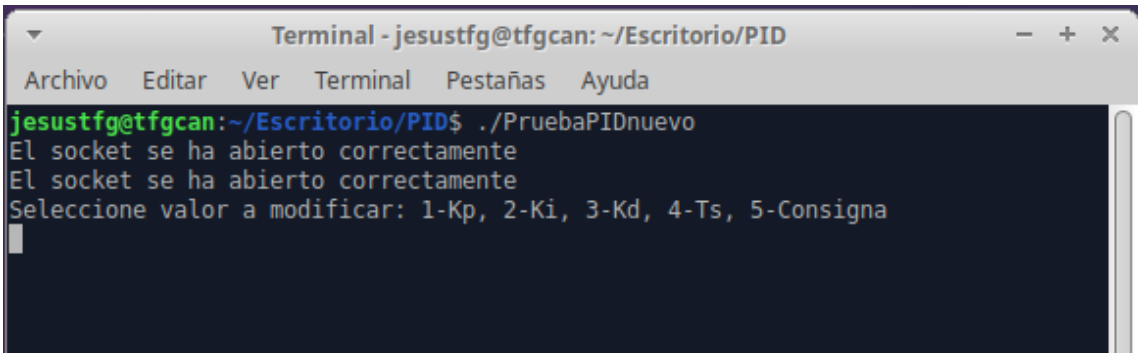

Figura 3.25. Ejecución prueba Control PID en PC

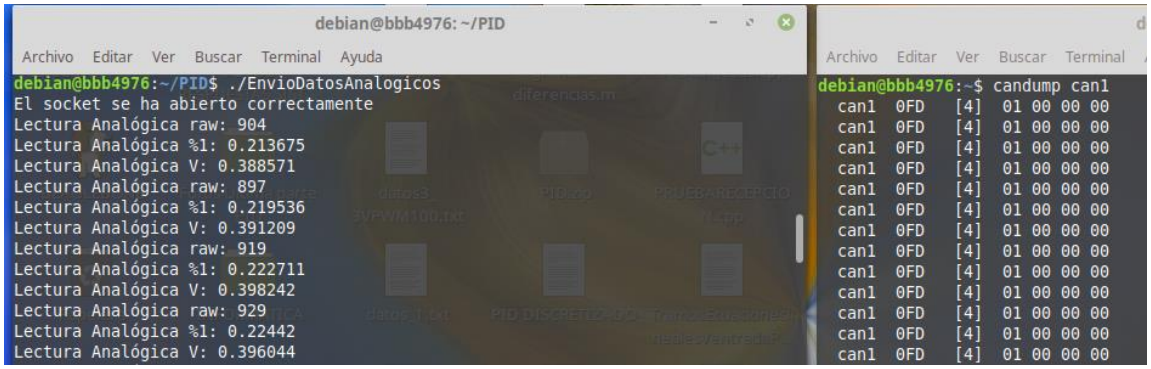

Figura 3.26. Ejecución de envío de datos analógicos en BBB y candump

En el PC, como se observa en la figura 3.25, se ejecuta el programa que implementa el control PID, donde se establecen los parámetros del controlador y se realiza la regulación de la salida en función de la consigna deseada.

Por otro lado, en la BBB se ejecuta el programa que se encarga de leer los datos analógicos y enviarlos a través del bus CAN. Estos datos analógicos representan la variable de entrada del sistema controlado por el control PID. Además, se utiliza el comando candump en la BBB para recibir las tramas que se envían desde el PC a través del bus CAN. Estas tramas contienen la consigna generada por el control PID.

De esta manera, la comunicación entre el PC y la BBB se establece mediante el bus CAN, donde el PC envía las consignas a la BBB a través de tramas CAN, y la BBB envía los datos analógicos al PC también mediante tramas CAN. Esto permite la interacción entre el control PID en el PC y el sistema controlado en la BBB, facilitando la regulación de la salida del sistema de acuerdo con la consigna establecida.

3.4.4 Recibir consigna en BeagleBone Black

En la última parte del sistema, de manera similar a la recepción de los datos analógicos, se implementará un programa que se encargará de recibir la consigna enviada desde el PC a través del bus CAN y aplicarla al control de la PWM.

Este programa se ejecutará en la BBB y estará a la espera de recibir las tramas CAN que contienen la consigna generada por el control PID en el PC. Una vez que se reciba la trama, se extraerá la consigna de la misma y se utilizará para ajustar la configuración de la PWM.

```
#include <iostream>
#include <iomanip>
#include <string>
#include "Frame.hpp"
#include "CAN.hpp"
#include "BBB_PWM.hpp"


int main() {
    // Crear objeto para elegir controlador
    CAN can(1);

    // Crear objeto para la clase PWM
    BBB_PWM pwm("P9_14");

    //Periodo pwm
    pwm.setPeriod(10000);
    pwm.setEnable(true);

    //Definir el filtro
    struct can_filter rfilter [1];
    rfilter[0].can_id = 0x253;
    rfilter[0].can_mask = CAN_SFF_MASK;

    setsockopt(can.getSocket(), SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));

    // Crear objeto para recibir trama
    Frame trama(can.getSocket());

    // Bucle para recibir tramas continuamente
    while (true) {
        // Recibir trama
        trama.recibirGuardarTrama();

        // Obtener los datos de la trama
        std::string tramaHex = trama.getdata();

        // Convertir la trama a un valor double
        double comando = std::stod(tramaHex, nullptr);

        //Establecer el comando en la PWM
        pwm.setDutyPer1(comando);
    }

    return 0;
}
```

Figura 3.27 Recepción de consigna en BBB

3.4.5. Prueba ejecutada

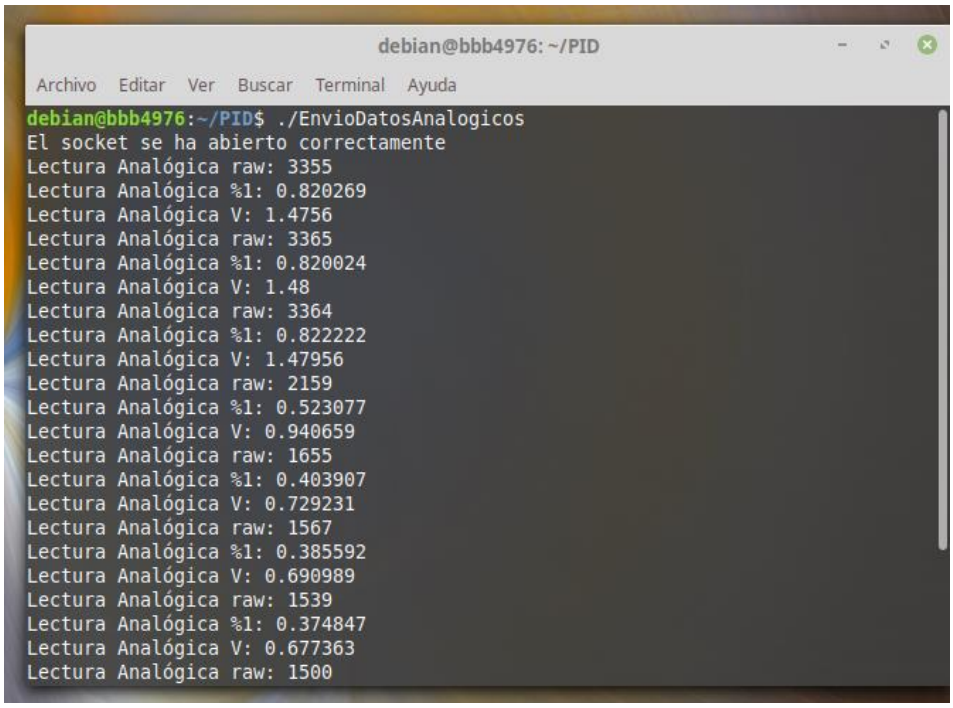

Figura 3.28. Envio de voltaje desde BBB a PC

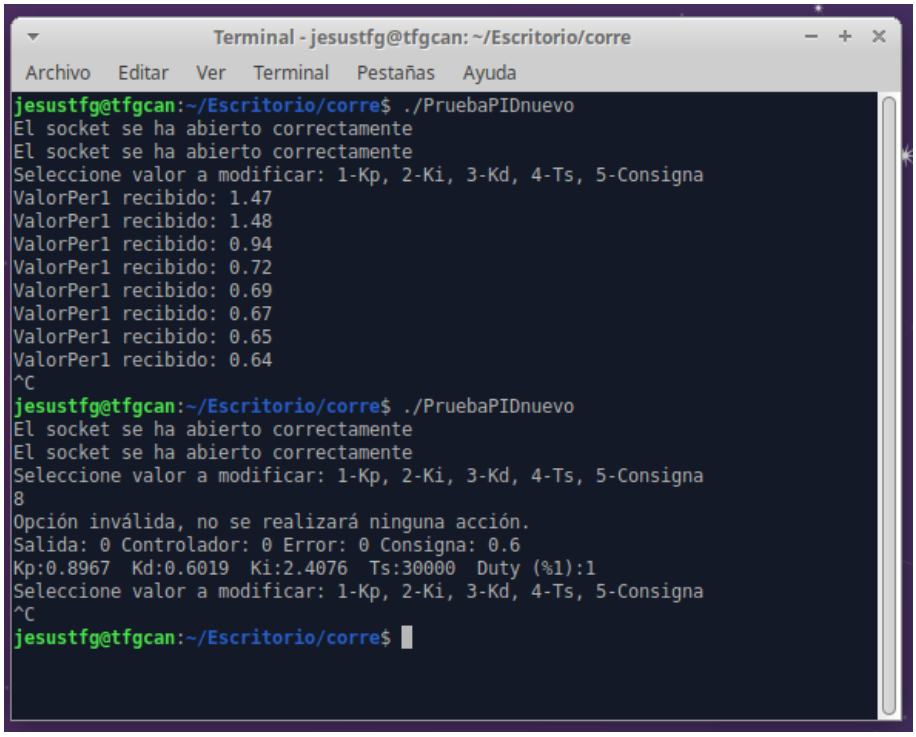

Figura 3.29 Ejecución de Prueba PID en PC

Capítulo 4. Conclusions and future work

The project focused on utilizing the CAN bus for communication and developing a control system. The CAN bus was used to transmit and receive data packets in Linux on a BeagleBone Black. Similarly, data packets were sent and received on a PC in Linux using CAN controllers.

CAN frames were successfully exchanged between the BeagleBone Black and the PC, establishing a communication link between the two devices. The programming language used for implementing the communication and control logic was C++.

To enhance communication efficiency, programs were developed to utilize filters on the CAN bus, allowing for selective reception based on frame identifiers.

Two essential classes were created: the CAN class, responsible for managing communication through CAN sockets in Linux, and the Frame class, designed for handling CAN frames. These classes facilitated sending, receiving, and manipulating CAN frames.

In addition to communication, programs were developed to make use of the ADC and PWM capabilities of the BeagleBone Black. This allowed for integration with devices requiring analog input or PWM control.

Furthermore, a temperature control system was implemented. C++ classes were developed to incorporate control algorithms such as ON-OFF and PID control. Communication between the BeagleBone and the PC was established, enabling data transfer for system control via the CAN bus. While de act (revisar) was doing in the BBB the control command was calculate on the PC.

In conclusion, this project successfully utilized the CAN bus for data transmission and reception between a BeagleBone Black and a PC running Linux. The developed C++ programs effectively managed communication. The integration of ADC and PWM capabilities expanded the range of compatible devices, while the temperature control system demonstrated the application of control algorithms in a practical scenery.

Future Work

Looking ahead, the project has future directions:

User Interface and Monitoring: Developing a user-friendly interface for easy configuration, monitoring, and centralized control of the system.

Enhanced Communication Protocols: Implementing advanced protocols like CAN-FD to increase data transfer rate and accommodate larger payloads for more complex applications.

Expansion to Industrial applications: Integrating with industrial CAN devices to enable seamless interface with industrial equipment and systems.

Exploring these areas will enable the project to evolve with improved functionality, scalability, and adaptability for various communication and device control applications.

Bibliografía

[1] Meinsa. (2022, 25 febrero). ¿Conoces la historia de las redes de comunicación industrial? Disponible: https://meinsa.com/2021/12/conoces-la-historia-de-las-redes-de-comunicacion-industrial/

[2] Aula21. (2022, 16 diciembre). Qué es un Bus de Campo y para qué sirve. Formación para la Industria. Disponible: https://www.cursosaula21.com/que-es-un-bus-de-campo/

[3] Solano, J. (s. f.). El modelo OSI. Disponible: http://dis.um.es/%7Elopezquesada/documentos/IES_1213/LMSGI/curso/xhtml/xhtml22/index.html

[4] Modelo OSI - Concepto, cómo funciona, para qué sirve y capas. (s. f.). Disponible: Concepto. https://concepto.de/modelo-osi/

[5] Javier. (2015, 10 septiembre). CAN BUS. Disponible: http://javiermk.blogspot.com/2015/09/can-bus.html

[6] Colaboradores de Wikipedia. (2022, 22 abril). Bus CAN. Wikipedia, la enciclopedia libre. Disponible: https://es.wikipedia.org/wiki/Bus_CAN

[7] CAN high / CAN low. (s. f.). Disponible: https://support.squarell.com/index.php?/Knowledgebase/Article/View/94/0/can-high--can-low

[8] SocketCAN. Controller Area Network. The Linux Kernel documentation. (s. f.). Disponible: https://www.kernel.org/doc/html/latest/networking/can.html

[9] BeagleBoard.org - bone101. (2018, 10 julio). Disponible: https://beagleboard.org/support/bone101

[10] Agarwal, T. (2021, 1 octubre). Beaglebone Black Microcontroller Datasheet: Working & Its Applications. ElProCus - Electronic Projects for Engineering Students. Disponible: https://www.elprocus.com/beaglebone-black-microcontroller/

[11] Waveshare SN65HVD230 Amazon.es. (s. f.). Disponible: https://www.amazon.es/SN65HVD230-CAN-Board-Communication-Development/dp/B00KM6XMXO

[12] SocketCAN - Controller Area Network — The Linux Kernel documentation. Disponible: (s. f.-b). https://docs.kernel.org/networking/can.html

[13] EMS Dr. Thomas Wünsche e.K. (2022, 10 noviembre). CAN Plug-In Board CPC-PCI. EMS Dr. Thomas Wünsche. Disponible: https://www.ems-wuensche.com/products/can-plug-in-board-cpc-pci/