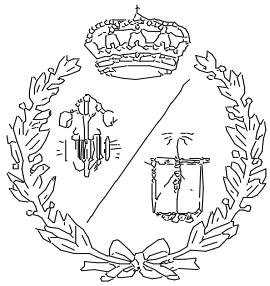
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIAse



Proyecto Fin de Máster

Estudio de la funcionalidad, módulos integrados y aplicaciones prácticas de CANalyzer

(Study of the functionality, integrated modules and practical applications of CANalyzer)

Para acceder al Título de

MÁSTER UNIVERSITARIO EN INGENIERÍA INDUSTRIAL

Autor: Rubén Rodríguez Llanes

Julio–2024

Resumen

TÍTULO	**Estudio de la funcionalidad, módulos integrados y aplicaciones prácticas de CANalyzer**		
AUTOR	**Rubén Rodríguez Llanes**		
DIRECTOR / PONENTE	**Fernando Fadón Salazar**		
TITULACIÓN	*Máster Universitario en Ingeniería Industrial*	**FECHA**	20/05/2024

PALABRAS CLAVE

CANalyzer, Panel Designer, CANdb+, CAPL, CANcase, MUXlab, base de datos, mensajería, ldf, bdc, trama, señal, variable de sistema, entorno, protocolo de comunicación, CAN, LIN, temporizadores, caja de cambios, calculador, ECU, bomba de aceite, marcha engranada, régimen de rotación, rpm, botón, código, script.

PLANTEAMIENTO DEL PROBLEMA

Desarrollar una interfaz de usuario para pilotar ciertos parámetros de dos calculadores del vehículo, verificando el correcto funcionamiento de los mismos, y analizar el tráfico de información entre un modelo simulink cargado en una MUXlab y los dos calculadores integrados en el vehículo, comprobando su implementación.

DESCRIPCIÓN DEL PROYECTO

Definición y análisis del funcionamiento y características propias del software industrial CANalyzer, así como sus diversos módulos integrados, tales como CAPL browser, Panel Designer y CANdb+. Aplicaremos este software para la resolución de dos casos de estudio, poniendo en práctica las características y módulos del mismo con el fin de demostrar las ventajas que ofrece CANalyzer.

CONCLUSIONES

He podido diseñar e implementar con éxito una interfaz de usuario en CANalyzer así como analizar el tráfico de información del vehículo satisfactoriamente, todo con un modesto presupuesto de menos de 16.300€. Así, queda demostrada la eficacia de CANalyzer como software de diagnóstico de datos y de diseño de elementos gráficos.

Este documento respeta la confidencialidad con Renault

BIBLIOGRAFÍA

https://es.wikipedia.org/wiki/Bus_CAN [1]

https://en.wikipedia.org/wiki/Local_Interconnect_Network [2]

https://www.vector.com/us/en/know-how/capl/#c221231 [3]

https://www.vector.com/int/en/events/global-de-en/webinar-recordings/2023/introduction-to-communication-access-programming-language-capl/ [4]

https://www.vector.com/es/es/support-downloads/support/#c10354 [5]

https://es.rs-online.com/web/ [6]

https://www.vector.com/int/en/events/global-de-en/webinar-recordings/2021/capl-fundamentals/ [7]

Índice General

DOCUMENTO Nº1: MEMORIA

APARTADO 1: INTRODUCCIÓN

1 Caso de estudio: Pilotado de ECUs mediante interfaz de usuario

1.1 Datos de partida proporcionados por el cliente

Un cliente nos ha encargado **diseñar una interfaz de usuario**, integrada en CANalyzer, que le permita pilotar y verificar ciertos parámetros de dos calculadores de un vehículo híbrido: poder **engranar la marcha del coche** mediante un calculador de la caja de cambios automática y **regular el régimen de rotación de una bomba de aceite.**

1.2 Solución adoptada

Para abordar la petición, usaremos dos módulos integrados en CANalyzer: **Panel Designer**, para diseñar la interfaz con los elementos necesarios, y **CAPL Browser,** para redactar scripts asociados a la interfaz que enviarán las consignas de usuario a los calculadores. El programa CANalyzer intercambiará señales con el calculador de la caja de cambios mediante un **bus CAN** y se comunicará con la bomba de aceite mediante un **bus LIN**.

2 Caso de estudio: Diagnóstico de tráfico de datos

2.1 Datos de partida proporcionados por el cliente

Por otra parte, el cliente nos ha aportado un modelo de Simulink completo para implementarlo en una MUXlab, conectar la misma a un coche y luego, mediante CANalyzer, **estudiar el tráfico de datos** entre la MUXlab, las ECUs implementadas en el vehículo y el banco.

2.2 Solución adoptada

Para abarcar el problema, tras haber realizado una prueba en coche, mediante CANalyzer analizaremos las **consultas** enviadas por el modelo Simulink a la toma OBD de un vehículo, después observamos las **respuestas** del vehículo al modelo Simulink y, finalmente, los **envíos** del modelo al banco de ensayos.

3 Planificación

A continuación, describo las fases que conllevará la realización de este proyecto.

Primero, tengo que realizar unos **estudios previos** del **hardware** con el que voy a trabajar, principalmente, la caja de cambios, las ECUs, la MUXlab y el cableado, así como el **software CANalyzer** y sus módulos integrados, conocer sus características, cómo funciona, los servicios que ofrece y cómo puedo emplearlo.

Después, en cuanto la interfaz de usuario, tengo que realizar un **diseño inicial**, que sea eficiente y práctico e implementar dicho diseño en CANalyzer, escribiendo unos scripts que ejecuten las acciones deseadas. Luego, realizaré **pruebas** para poner en práctica el diseño e iterar hasta que haya localizado y resuelto cualquier posible error de programación o de cableado.

Por otra parte, en relación con el análisis de tráfico de señales, realizaré una **prueba en un prototipo de vehículo** aportado por la empresa y, una vez realizado, **analizaré los resultados en CANalyzer** y en base a ellos sacaré conclusiones.

APARTADO 2: ESTUDIOS Y ANÁLISIS PREVIOS

1 Hardware

Es imperativo conocer los componentes más relevantes del **vehículo** para poder llevar a cabo nuestro proyecto. Para empezar, partimos del conjunto **GMP (Grupo MotoPropulsor) o Powertrain,** que engloba todos los componentes que generan potencia para poder desplazar el vehículo. Esto incluye el ICE (motor de combustión interna), la transmisión, la batería y la caja de cambios. En el caso de coches híbridos, el GMP también incluye uno o varios motores eléctricos. Para nuestra demanda, nos concentraremos en la **caja de cambios**.

Figura 1. Powertrain de un vehículo

Una caja de cambios puede ser automática o manual. En esta petición, tratamos con una **caja de cambios automática**. Una caja automática puede cambiar por sí misma la relación de marcha de forma automática según el vehículo se desplace, de manera que el conductor no tendrá que cambiar de marcha él mismo.

Figura 2. Caja de cambios automática

La caja de cambios contiene varias **ECUs** (Electronic Control Unit) **o calculadores**, dispositivos electrónicos conectados a sensores que les permiten enviar y recibir señales, que pilotan actuadores y relés que ejecutan comandos al recibir dichas señales. En nuestro caso, trataremos con dos calculadores:

- Un **calculador de la caja de cambios** que se encarga de engranar las relaciones de marcha en la caja.

- Una **bomba de aceite** que proporciona un caudal de aceite a la caja de cambios que lubrica los engranes y otros componentes.

Ambos calculadores, en condiciones normales, intercambiarían información con la **ECU principal del powertrain** dentro del vehículo. Mediante el programa CANalyzer y la interfaz de usuario que diseñaremos posteriormente, **nuestro ordenador actuará como dicha ECU**, enviando y recibiendo señales al calculador y la bomba.

Para que CANalyzer pueda enviar, recibir e interpretar información, es necesario conectar nuestro ordenador a dichos calculadores. Sin embargo, tanto el ordenador como las ECUs utilizan **protocolos de comunicación diferentes**, luego el intercambio de información no sería posible directamente. Por eso, es necesario usar una **CANcase**, una interfaz de red CAN/LIN desarrollada por la compañía alemana Vector Informatik GmbH, capaz de traducir la información procedente del ordenador a protocolos CAN y LIN y viceversa. Así, podremos establecer la comunicación entre nuestro ordenador y los calculadores.

Figura 3. Tipos de CANcase

La CANcase posee varios canales de bus, a los cuales podemos conectar elementos como ECUs o MUXlabs, y un canal USB tipo B, al cual conectamos nuestro ordenador. Por otra parte, hay varios modelos de CANcase, como la **VN1630A**, de pequeño tamaño, con 2 canales bus y que no necesita alimentación externa, o la **VN5620**, de mayor tamaño, con 4 canales bus y que sí requiere de alimentación externa. En nuestro proyecto, usaremos la CANcase VN1630A.

Luego, tendremos una **fuente de alimentación** que aporta tensión al calculador y la bomba de aceite para habilitar su funcionamiento.

Por otra parte, para poder analizar el tráfico de datos, hay que considerar los **bancos de ensayo**, plataformas para experimentación de proyectos de gran desarrollo, tales como el GMP. Los bancos de pruebas brindan una forma de comprobación rigurosa, transparente y repetible de la información pertinente a componentes del vehículo.

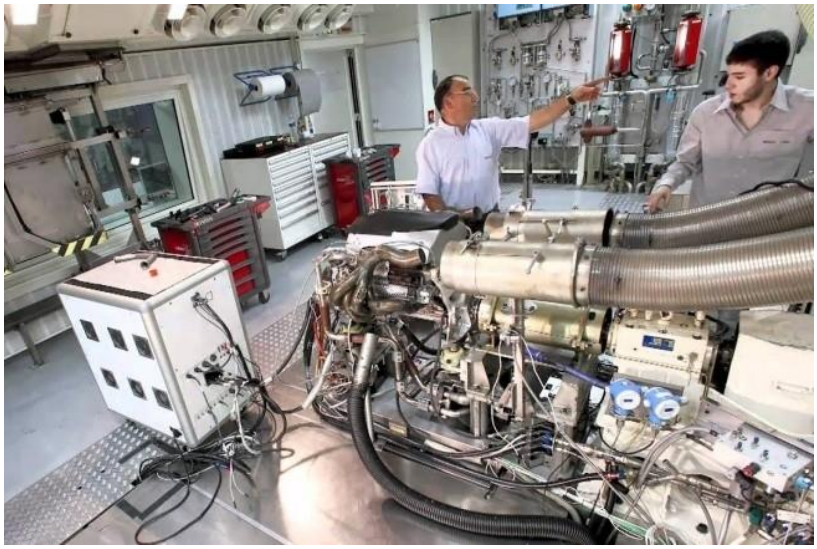


Figura 4. Banco de ensayos.

Para afrontar este caso de estudio, utilizaremos una **MUXlab**. Se trata de un dispositivo desarrollado por la compañía **Capgemini**, anteriormente Altran, y en el cual se puede cargar un modelo diseñado en Simulink, un módulo integrado del programa MATLAB. En nuestro caso de estudio, la MUXlab asume el rol de una **herramienta de diagnóstico** durante la prueba:

- El modelo cargado en la MUXlab enviará **consultas** a dos ECUs del vehículo, la caja de cambios y la bomba de aceite, a través de la toma OBD (On-Board Diagnostic) del coche. Las consultas solicitarán comprobar determinados parámetros de los calculadores, específicamente la marcha engranada en la caja y el régimen de giro de la bomba.

- Consecuentemente, el modelo recibirá **respuestas** de los calculadores, indicando valores instantáneos de cada parámetro que ha sido solicitado.

- Finalmente, el modelo realizará **envíos** al banco de ensayos, para poder procesar y visualizar la información adecuadamente.

Figura 5. Dispositivo MUXlab6.

Para el proyecto, utilizaré una **Muxlab6**, que dispone de 6 puertos CAN, 2 puertos LIN y 4 puertos SENT. Al comprar una MUXlab, Capgemini también proporciona software adicional relacionado con la MUXlab, como la **biblioteca MUXlink** para Simulink, una biblioteca de elementos que permite a los modelos enviar y recibir señales a calculadores, y **USB Firmware Updater**, que nos permitirá cargar el modelo Simulink en la MUXlab.

Finalmente, tendremos en cuenta el **cableado** necesario para llevar a cabo ambos casos de estudio. Para la interfaz de usuario, conectaremos el ordenador con la CANcase, la CANcase con las ECUs y las ECUs con la fuente de alimentación (véase fig. 55 y 56). Para el diagnóstico de tráfico de datos, conectaremos el ordenador con la CANcase, la CANcase con la toma OBD del coche, la CANcase con la MUXlab preparada y la MUXlab con el mechero del coche (véase fig. 59).

2 Buses y protocolos de comunicación

Un **bus o canal** es un sistema digital que transfiere datos entre dos ECUs de un vehículo. Esta expresión cubre todos los componentes de hardware y software relacionados, englobando los **protocolos de comunicación**.

El **protocolo de comunicaciones** es un sistema de reglas que permite que dos o más entidades de un sistema de comunicación se comuniquen entre sí para transmitir información por medio de cualquier tipo de variación de una magnitud física. Es decir, se trata de las reglas o el estándar que define la sintaxis, semántica y sincronización de la comunicación, así como posibles métodos de recuperación de errores.

Los protocolos se pueden clasificar según el **modelo TCP/IP**, un modelo de referencia de la IETF (Internet Engineering Task Force), la organización internacional abierta de normalización. Este modelo se divide en 4 capas, de menor a mayor: Acceso al medio, Internet, Transporte y Aplicación.

Los protocolos también se pueden clasificar por el **modelo OSI**, creado por la Organización Internacional de Normalización (ISO). Este modelo se divide en 7 capas, de menor a mayor: Física, Enlace de datos, Red, Transporte, Sesión, Presentación y Aplicación.

Figura 6. Comparación modelo OSI vs. Modelo TCP/IP

2.1 Protocolo CAN

CAN (Controller Area Network) es un protocolo de comunicaciones desarrollado por la firma alemana Robert Bosch GmbH, basado en una topología bus para la transmisión de mensajes en entornos distribuidos. Además, ofrece una solución a la gestión de la comunicación entre múltiples CPUs. [1]

El protocolo de comunicaciones CAN proporciona los siguientes **beneficios**:

- Aporta la habilidad para el **autodiagnóstico**, una elevada inmunidad a las interferencias y la reparación de errores de datos.

- Es **normalizado**, es decir, simplifica y economiza la tarea de comunicar subsistemas de diferentes fabricantes sobre un bus común.

- El procesador anfitrión delega la carga de comunicaciones a un periférico inteligente, por lo que el procesador anfitrión dispone de más tiempo para ejecutar sus propias tareas.

- Al tratarse de una red multiplexada, disminuye notablemente el cableado y elimina las conexiones punto a punto, excepto en los enganches.

El protocolo CAN tiene **dos capas**, una física y otra de enlace de datos estandarizadas, luego pertenecería a los protocolos de **Acceso al medio** según el modelo TCP/IP. Está compuesto por dos cables trenzados, CAN_L (CAN low) y CAN_H (CAN high). Según la tasa de bits, tenemos varios tipos de protocolo CAN:

- **CAN HS**: CAN High Speed. Tiene una resistencia de 120 Ω. Según su tasa de bits, este protocolo a su vez se divide en:

    - **CAN 500**: Tiene una tasa de bits de 500 kbps. Tiene un DLC de 8 bytes.

    - **CAN FD**: CAN Flexible Data-rate. Puede tener tasas de bits de 5Mbps o de 8Mbps. Es compatible hacia atrás, es decir, puede leer mensajes de CAN clásico o de CAN 2.0. Tiene un DLC de 64 bytes.

- **CAN LS**: CAN Low Speed tolerante de fallos. Tiene una tasa de bits de hasta 125 Kbps. Tiene una resistencia de 100 Ω.

2.2 Protocolo LIN

LIN (Local Interconnect Newtork) es un protocolo de comunicación serial que apoya comunicaciones de hasta 19.2 Kbit/s. Es un sistema lento y pequeño que se utiliza como una sub-red barata del bus CAN para integrar los dispositivos o actuadores inteligentes en los coches actuales. [2]

Algunas de las **ventajas** aportadas por el bus LIN son la facilidad de manejo, la disponibilidad de componentes, ser más barato que CAN y otros buses de comunicación, no requiere de licencia de protocolo y su extensión es fácil de implementar.

3 Arquitectura eléctrico-electrónica de nuestro proyecto

En las redes de comunicación, tenemos **señales,** notificaciones asíncronas enviadas a procesos para informarles de eventos. Cuando se envía una señal a un proceso, el sistema operativo modificará la ejecución normal de dicho proceso.

Por otra parte, tenemos **mensajerías** o **bases de datos,** documentos asociados a buses que codificarán las señales intercambiadas entre las dos ECUs conectadas al bus. El formato de la base de datos empleada depende del **protocolo de comunicación** utilizado en los buses. Por ejemplo, las señales de protocolo CAN se codificarán mediante archivos .dbc, mientras que las señales de protocolo LIN emplearán archivos .ldf.

Dado un bus **LIN**, desde el .ldf asignamos **nodos** a las ECUs conectadas al bus según la función que desempeñen en el tráfico de datos. Distinguimos dos tipos de nodos, el nodo **Master**, que manda señales de consigna al nodo **Slave**, el cual consecuentemente realizará determinados procesos para satisfacer dichas demandas. Luego, el nodo Slave enviará señales de respuesta al nodo Master para mostrar los resultados obtenidos tras el proceso y, en condiciones ideales, verificar el correcto funcionamiento de las ECUs.

Un mismo .ldf puede tener **múltiples nodos de esclavo**, de manera que un mismo .ldf puede ser compatible para **múltiples buses**, entre la ECU maestra y cualquiera de las ECUs esclavas. También sería posible implementar un sistema bus que permita conectar una ECU maestra a **dos o más ECUs esclavas simultáneamente**.

Por otra parte, dado un bus **CAN**, desde el .dbc también asignamos **nodos** a las ECUs. Sin embargo, en este caso no se describe la comunicación entre varias ECUs esclavas y una sola ECU maestra, sino que tienen lugar diálogos más complejos entre múltiples ECUs, cediendo la palabra unas a otras consecutivamente. Por esa razón, se suele reservar el protocolo LIN para calculadores más sencillos, como la bomba de aceite, mientras que se utiliza el protocolo CAN para la gran mayoría de calculadores de un vehículo, que poseen una lógica interna mucho más compleja.

A nivel estructural, las bases de datos se dividirán en **tramas.** La trama es la unidad de envío de datos, una serie sucesiva de bits organizados de forma cíclica que transportan señales y que permiten extraer la información dentro de dichas señales en la recepción. Vendría a ser análogo al Paquete de red en el Nivel de red del modelo OSI.

Por lo general, la trama estará constituida por una **cola, datos y cabecera**. El chequeo de errores CRC suele estar en la cola y los campos de control de protocolo estarán en la cabecera, mientras que la parte de datos es la que transmitirá en nivel de comunicación superior, típicamente el Nivel de red.

Desde el punto de vista de CANalyzer, podemos distinguir entre **tramas RX**, cuyas señales se envían al ECU de interés a través del bus, y **tramas TX**, cuyas señales son recibidas desde el ECU mediante el bus.

3.1 Pilotado de ECUs mediante interfaz de usuario

El cliente nos aportó documentación adicional relacionada a otros proyectos donde se han utilizado las mismas bases de datos que usaremos en nuestra petición, **DatabaseGearbox.dbc y DatabasePump.ldf**. Estudiamos dicha documentación con el fin de comprender mejor la arquitectura de dichas bases de datos, identificar las tramas y señales relevantes de cara a nuestro proyecto y entender el funcionamiento y propósito de las mismas.

3.1.1 DatabaseGearbox.dbc

Esta mensajería se corresponde al bus CAN del calculador de la caja de cambios automática. El nodo master corresponde a la ECU principal del powertrain y el nodo Slave corresponde al calculador de la caja. Nos interesa trabajar con dos tramas de esta base de datos:

- **MessageRXCAN**. Se trata de una trama RX cuyas señales enviamos al calculador. Trabajaremos con las siguientes señales:
    - **ModeDemand**. Nos permite seleccionar el modo de operación del motor. Según el valor asignado:
        - Si se da el valor "0", entonces la ECU **no ha recibido petición**, luego la señal estaría inactiva.
        - Si se da el valor "1", entonces el motor adopta el modo de "**control de las posiciones**", permitiendo la regulación del motor. Este es el modo de operación en el que nos interesa trabajar.
        - Si se da el valor "2", entonces el motor adopta el modo de "**control de las corrientes**", es decir, el modo aprendizaje por corriente.

      Comprobando la señal en la documentación interna de referencia, confirmamos los efectos de los valores de la misma.
    - **GearDemand.** Nos permite escoger la marcha de la caja de cambios automática que queremos engranar.
- **MessageTXCAN**. Se trata de una trama TX cuyas señales recibimos desde el calculador. Trabajaremos con la siguiente señal:
    - **CurrentGear.** Indica la marcha engranada actualmente.

3.1.2 DatabasePump.ldf

Esta mensajería se corresponde al bus LIN de la bomba de aceite. El nodo master corresponde a la ECU principal del powertrain y el nodo Slave corresponde a la bomba de aceite. Nos interesa trabajar con dos tramas de esta base de datos:

- **MessageRXLIN**. Trama RX cuyas señales enviamos a la bomba de aceite. Trabajaremos con la siguiente señal:
    - **RotationDemand**. Nos permite seleccionar el régimen de rotación deseado de la bomba de aceite en rpm.
- **MessageTXLIN**. Trama TX cuyas señales recibimos de la bomba de aceite. Trabajaremos con la siguiente señal:
    - **CurrentRotation**. Nos devuelve el régimen de rotación real de la bomba en rpm.

3.2 Diagnóstico de tráfico de datos

El cliente nos aportó dos bases de datos, **DIAGNOSTIC_TOOL_TO_OBD.dbc,** para codificar las consultas del modelo a las ECUs del coche y las respuestas de las ECUs al modelo, y **DIAGNOSTIC_TOOL_TO_BENCH.dbc**, para codificar los envíos del modelo al banco.

3.2.1 DIAGNOSTIC_TOOL_TO_OBD.dbc

Esta mensajería corresponde al bus CAN conectado a la **toma OBD del vehículo**. Contiene una trama de **consulta**, enviada por el modelo Simulink, y varias tramas de **respuesta**, enviadas por los calculadores del vehículo al recibir consultas. Nos fijaremos en tres tramas:

- **DIAGNOSTIC_TOOL_to_ALL_ECUs**. Trama que contiene las **consultas** del modelo Simulink, incluyendo la **marcha engranada** y el **régimen de rotación de la bomba actual**. Es transmitida por el modelo Simulink y recibida por las ECUs del coche, incluyendo el calculador de la caja de cambios y la bomba de aceite.

- **PUMP_to_DIAGNOSTIC_TOOL**. Trama de **respuesta** a la consulta, transmitida por la bomba de aceite y recibida por el modelo Simulink.

- **GEARBOX_to_DIAGNOSTIC_TOOL.** Trama de **respuesta** a la consulta, transmitida por el calculador de la caja de cambios y recibida por el modelo Simulink.

3.2.2 DIAGNOSTIC_TOOL_TO_BENCH.dbc

Esta mensajería corresponde al bus CAN conectado al banco de ensayo, contiene tramas de **envío,** correspondientes a las consultas del modelo Simulink, que serán recibidas por el banco.

- **DELIVERY_GEAR**. Trama de **envío**, contiene la marcha engranada, transmitida por el modelo Simulink y recibida por el banco de ensayos.

- **DELIVERY_ROTATION**. Trama de **envío**, contiene el régimen de rotación de la bomba actual, transmitida por el modelo Simulink y recibida por el banco de ensayos.

4 CANalyzer

CANalyzer es una herramienta de software desarrollada por la compañía alemana Vector Informatik GmbH, principalmente usada por los suministradores de ECUs para analizar el tráfico de datos en sistemas bus, tales como CAN, LIN, FlexRay, Ethernet y MOST, la edición de scripts y el diseño de elementos gráficos.

A continuación, explicaré las opciones y características más importantes de CANalyzer para poder llevar a cabo nuestro proyecto.

4.1 File

En primer lugar, es posible **guardar la configuración actual de CANalyzer** en un archivo .cfg, de forma que, al abrir dicho archivo, se restablecen todos los parámetros guardados en el mismo, como las bases de datos asociadas, modo de simulación online u offline, registros de simulaciones, etc. Basta con ir a *File* y seleccionar ***Save* o *Save as*** para guardar la configuración en el directorio deseado. Después, hay que seleccionar ***Open*** para cargar la configuración deseada.

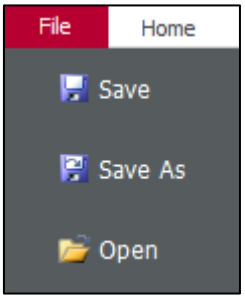


Figura 7. Guardado y carga de configuraciones

Cada vez que ejecutamos la simulación de un modelo, sobre todo si cambiamos de modelo, es importante tener en cuenta los **directorios** donde guardamos las configuraciones. Si no se tiene cuidado, podemos acabar sobreescribiendo archivos .cfg anteriores.

4.2 Home

Para inicializar una medición, pulsamos "*Start*", el botón con forma de trueno, y para detener la medición, pulsamos "*Stop*", el botón con forma de octógono rojo.

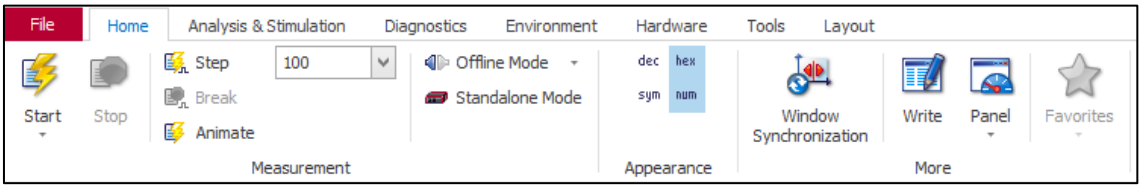


Figura 8. Opción Home

También podremos añadir paneles, diseñados previamente en el módulo integrado "*Panel Designer*", mediante el botón "*Panel*" y luego "*Add Panel*". Así añadiremos nuestra interfaz de usuario, Panel_TFM, a nuestra configuración de CANalyzer.

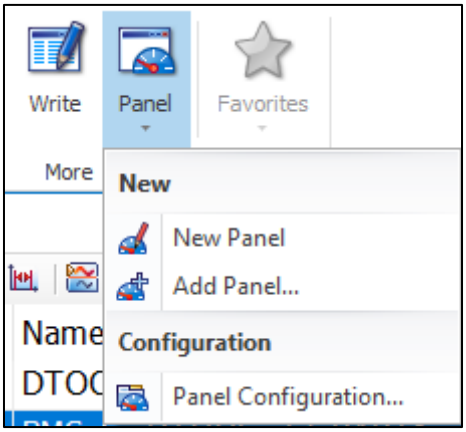


Figura 9. Añadir paneles

También podremos abrir la ventana "*Write*" que, durante la medición, nos devolverá mensajes de texto en función de las consignas que asignemos para indicar si las mismas son correctas o no.

4.3 Opción Configuration

4.3.1 Measurement Setup

Dentro de la opción "*Configuration*", tenemos "*Measurement Setup*", una ventana que nos muestra un esquema representativo de la simulación.

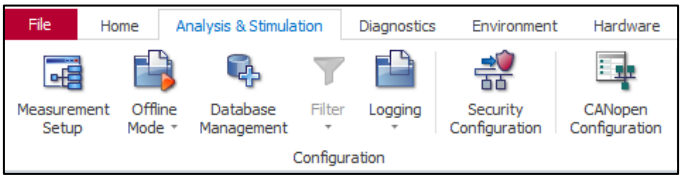


Figura 10. Opción Analysis & Stimulation

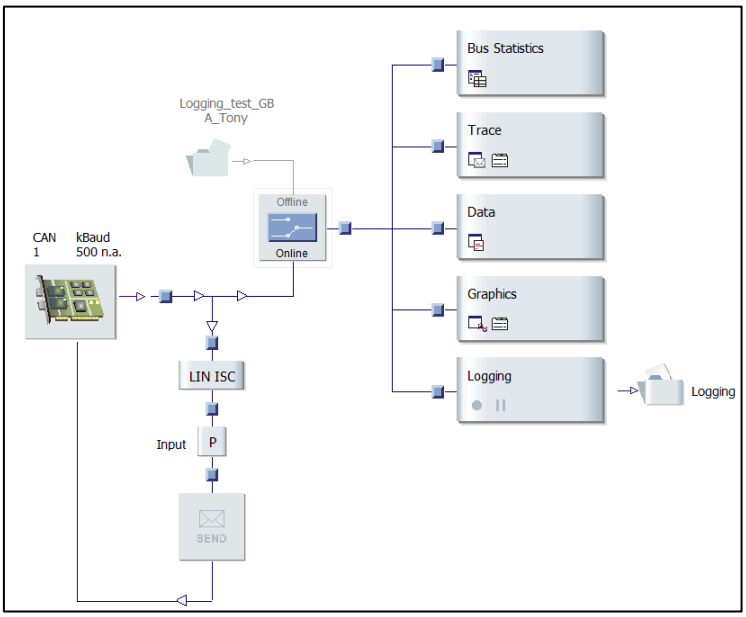


Figura 11. Ventana Measurement Setup.

Desde la ventana "*Measurement Setup*", podemos interactuar con el **switch,** que determinará el modo de la simulación, online u offline, el **nodo de programa,** que estará asociado a un script que organizará el tráfico de información con las ECUs, y los **hotspots**, unos cuadrados azules que, al hacer click sobre ellos, nos permitirán habilitar o inhabilitar determinados bloques en función de nuestras necesidades.

Entraré en más detalle sobre los **bloques de salida** (Trace, Graphics y Logging) y los modos **online/offline** más adelante.

4.3.2 Database Management

Otra opción disponible es **asociar bases de datos** (archivos .dbc, .ldf, etc) a los canales correspondientes de CANalyzer. En *Analysis & Simulation,* hay que ir a *Database Management*, hacer click derecho en el canal que proceda y luego en *Add database* (**véase fig. 10**).

Para la interfaz de usuario, asociamos la mensajería **DatabaseGearbox.dbc** al canal CAN1 de CANalyzer y la mensajería **DatabasePump.ldf** al canal LIN1 de CANalyzer.

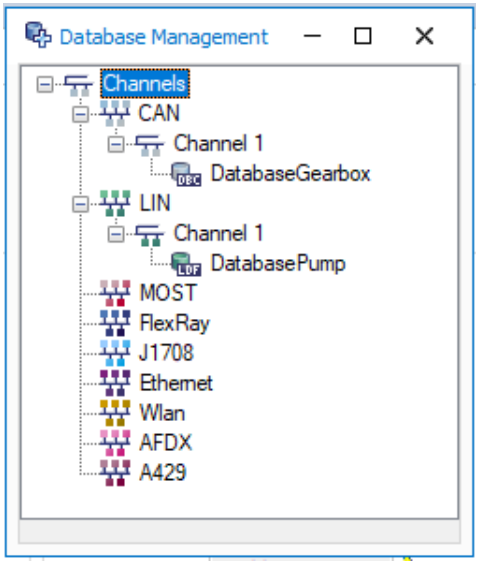


Figura 12. Mensajerías para la interfaz de usuario.

Para el diagnóstico de tráfico de datos, asociamos las mensajerías **DEBUG_TOOL_to_OBD.dbc** y **DEBUG_TOOL _to_BENCH.dbc** a los canales CAN1 y CAN2 de CANalyzer.

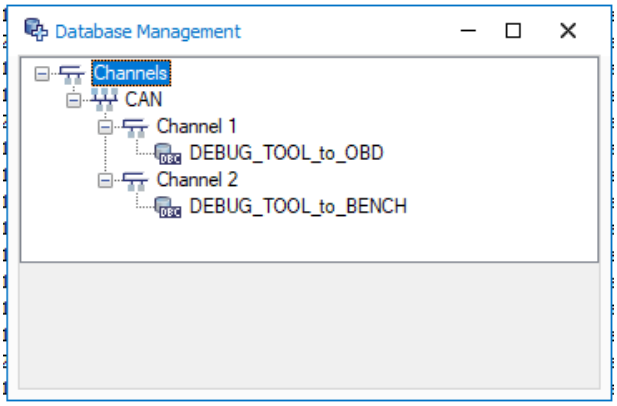


Figura 13. Mensajerías para el diagnóstico de tráfico de datos.

4.3.3 Logging

También es posible **registrar la simulación ejecutada y ejecutar dicho registro offline** para analizar resultados, sin estar conectados a ningún hardware. Para ello, pulsamos "*Logging*" y pulsamos "*Insert Logging Block*" para añadir un **bloque de registro** que figurará en la ventana "*Measurement Setup*" (**véase fig. 11**) y que realizará el registro deseado. Si se activa el tick en la casilla del bloque de registro, se **guardará un registro después de ejecutar cada simulación**.

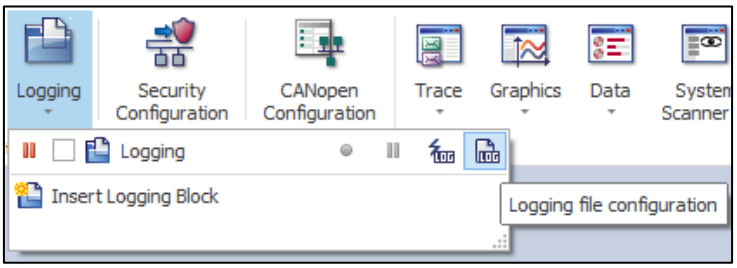


Figura 14. Logging

Después, elegimos Logging File Configuration para configurar dicho registro.

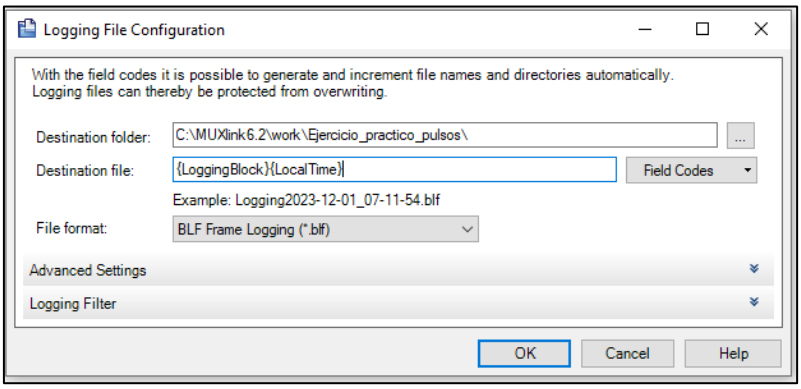


Figura 15. Configuración de registro

Podemos escoger el directorio, formato de archivo (.blf, .csv, etc) y el nombre del archivo, este último añadiendo campos tales como el nombre de usuario, tiempo local, versión del programa, etc.

Conviene **desactivar el bloque de registro** cuando no queramos guardar los resultados generados. Así evitaremos tener registros innecesarios y redundantes o guardar registros en la carpeta equivocada.

4.3.4 Modos de simulación

Es posible trabajar con CANalyzer o bien en **modo online**, conectado a las interfaces físicas, o bien en **modo offline**, ejecutando registros generados previamente sin estar conectado a ningún dispositivo hardware. Para pasar de modo online a offline y viceversa, basta con hacer doble click en el switch de la ventana *Measurement Setup* **(véase fig. 11)**.

- Si habilitamos el *modo online* de CANalyzer, podemos ejecutar simulaciones en tiempo real.

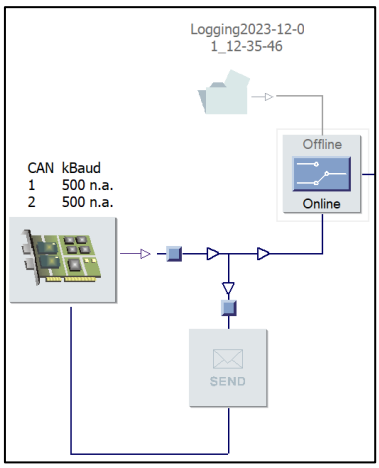


Figura 16. Modo Online.

- Si habilitamos el *modo offline* de CANalyzer, podemos cargar espontáneamente, no en tiempo real, simulaciones guardadas previamente en archivos .blf.

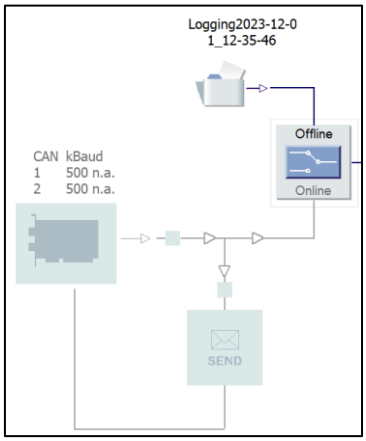


Figura 17. Modo Offline.

4.4 Opción Analysis

4.4.1 Trace

Para visualizar las señales de salida del modelo simulink **numéricamente**, vamos a *Analysis & Stimulation* y abrimos la ventana *Trace*.

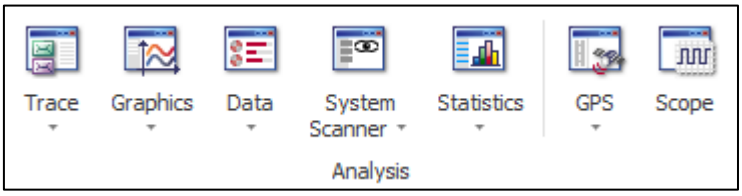


Figura 18. Opción Analysis.

Si pulsamos *Display Mode*  , podemos ver todas las muestras tomadas de las señales y el instante en el que se ha tomado cada una.

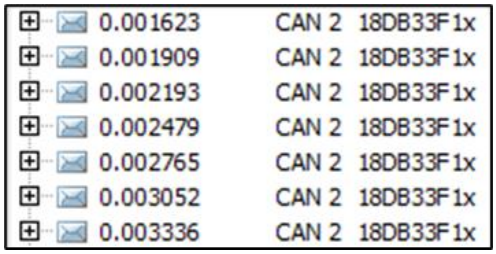


Figura 19. Display Mode.

Si pulsamos *Toggle Time Mode*  , podemos ver el tiempo que ha pasado entre muestra y muestra. Esta opción es útil para **verificar los períodos de las consultas** de Simulink, es decir, comprobar que las consultas se envían cada cierto intervalo de tiempo de manera constante.

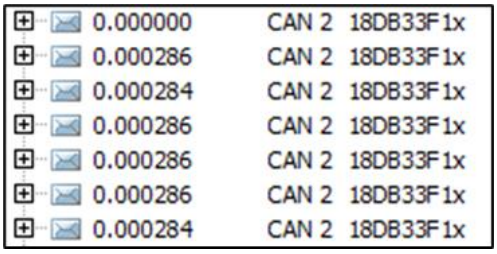


Figura 20. Toggle Time Mode.

4.4.2 Graphics

Para visualizar las señales de salida **gráficamente,** vamos a *Analysis & Stimulation* y abrimos la ventana *Graphics*. Para añadir las señales correspondientes a visualizar, vamos a *Graphics*, en la ventana de señales hacemos click derecho y pulsamos a *Add Signals*.

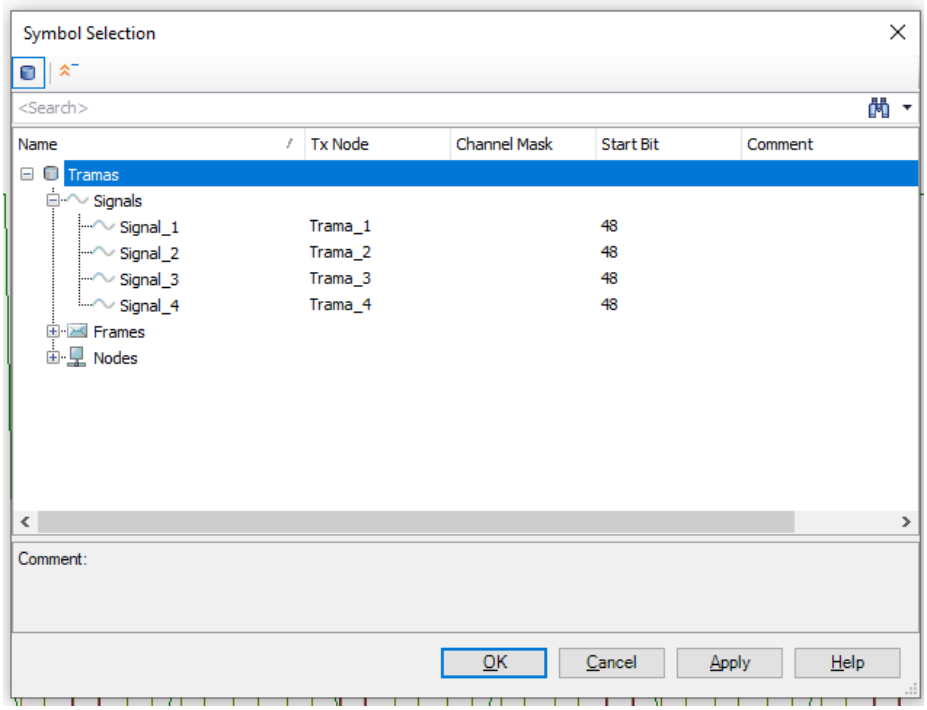


Figura 21. Selección de señales.

Por defecto, cada señal está asignada a un eje de ordenadas diferente, así que conviene crear un **eje Y común para todas**. Seleccionamos todas las señales, hacemos click derecho y seleccionamos *Create Common Axis*. Así, podremos ajustar el eje de ordenadas manualmente haciendo zoom con el ratón.

4.5 Opción Stimulation

En la opción *Stimulation*, añadimos el bloque *LIN Interactive Scheduler* en serie con el nodo de programa asociado a nuestro script y seleccionamos RUN_MAIN como tabla de inicio durante la inicialización de la medición. Esta tabla enviará las tramas RX del archivo ldf a la bomba.

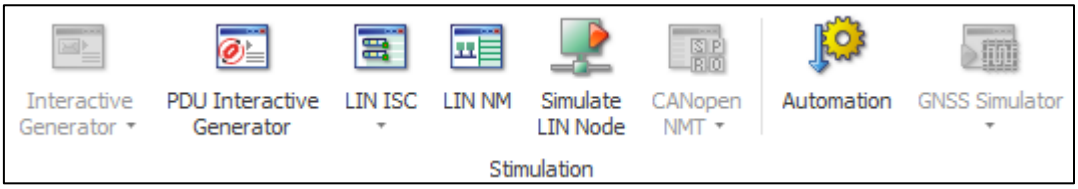


Figura 22. Opción Stimulation

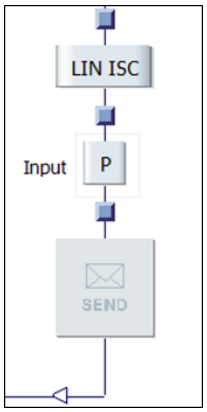


Figura 23. Opción Tools

4.6 Opción Environment

Yendo a "*Environment*" y luego a "*System Variables*", podemos **crear entornos con sus propias variables de sistema**. Las variables de sistema son variables de CANalyzer declaradas localmente que podrán ser definidas y manipuladas en scripts de CAPL y que podrán interactuar con las señales intercambiadas con las ECUs. Por ejemplo, podremos transferir valores de estas variables a señales de tramas RX que mandemos a las ECUs, o extraer y almacenar valores de señales de tramas TX que recibamos de las ECU.

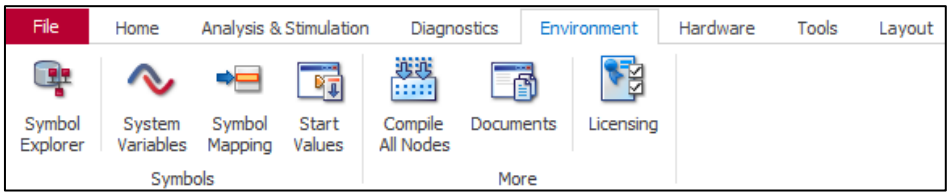


Figura 24. Opción Environment.

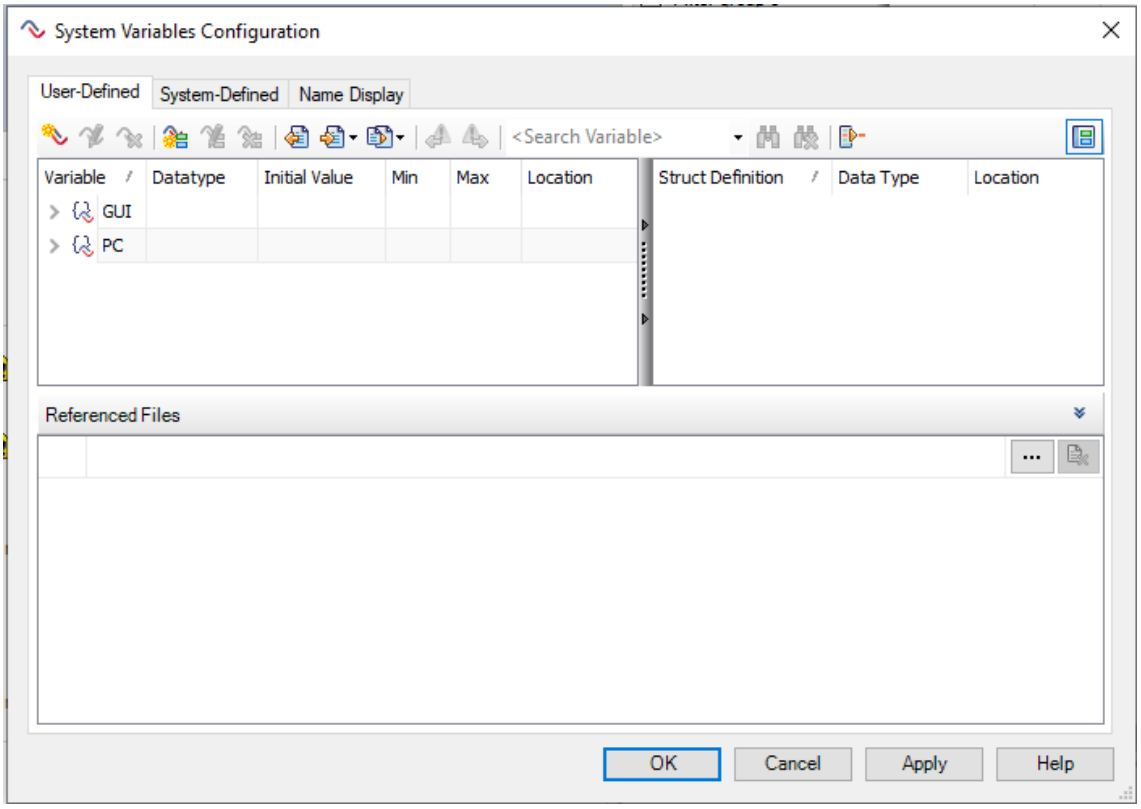


Figura 25. Configuración de variables de sistema.

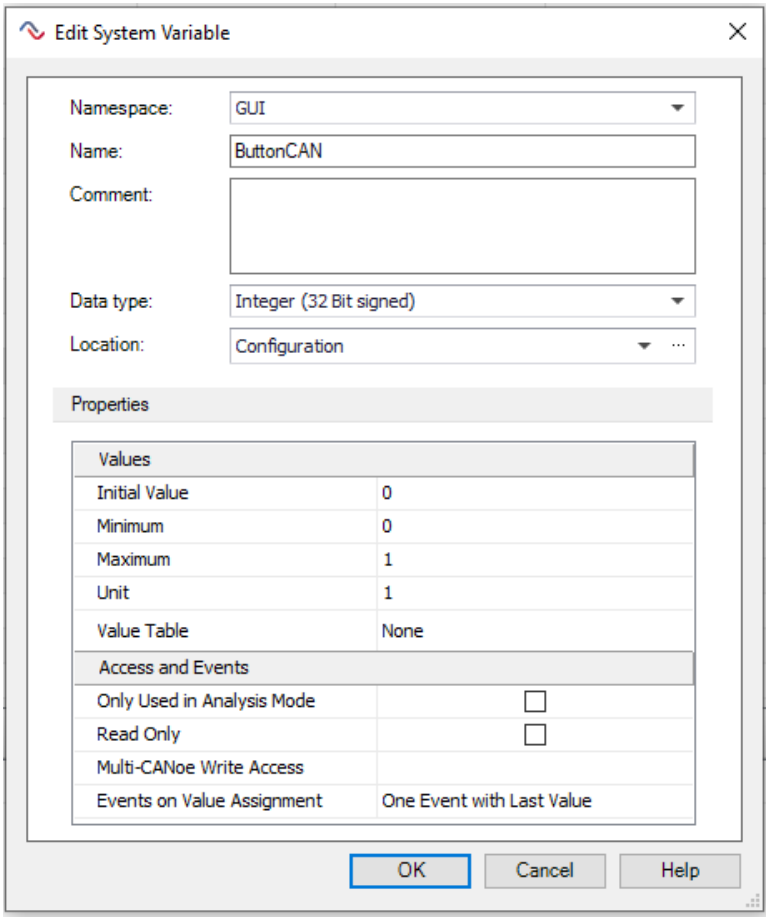


Figura 26. Variable de sistema declarada

Si bien no se pueden guardar entornos ni variables de sistema sin licencia CANalyzer, sí que es posible **exportar entornos** a un archivo .vsysvar externo, de manera que, al abrir una configuración nuevamente sin licencia, bastaría con importar dicho archivo .vsysvar para poder usar sus entornos y variables directamente, sin tener que definirlos otra vez.

4.7 Hardware

Por una parte, hay que **configurar los canales CAN y LIN** que vamos a usar. A esos efectos, vamos a "*Hardware*" y luego pulsamos "*Channel Usage*". Así, se abre un cuadro de diálogo donde podemos seleccionar el número de canales que queremos usar. Para este proyecto, establecemos 1 canal CAN y 1 canal LIN.

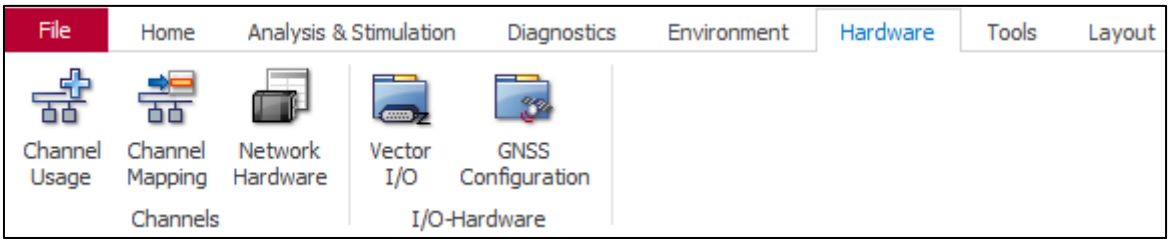


Figura 27. Opción Hardware de CANalyzer

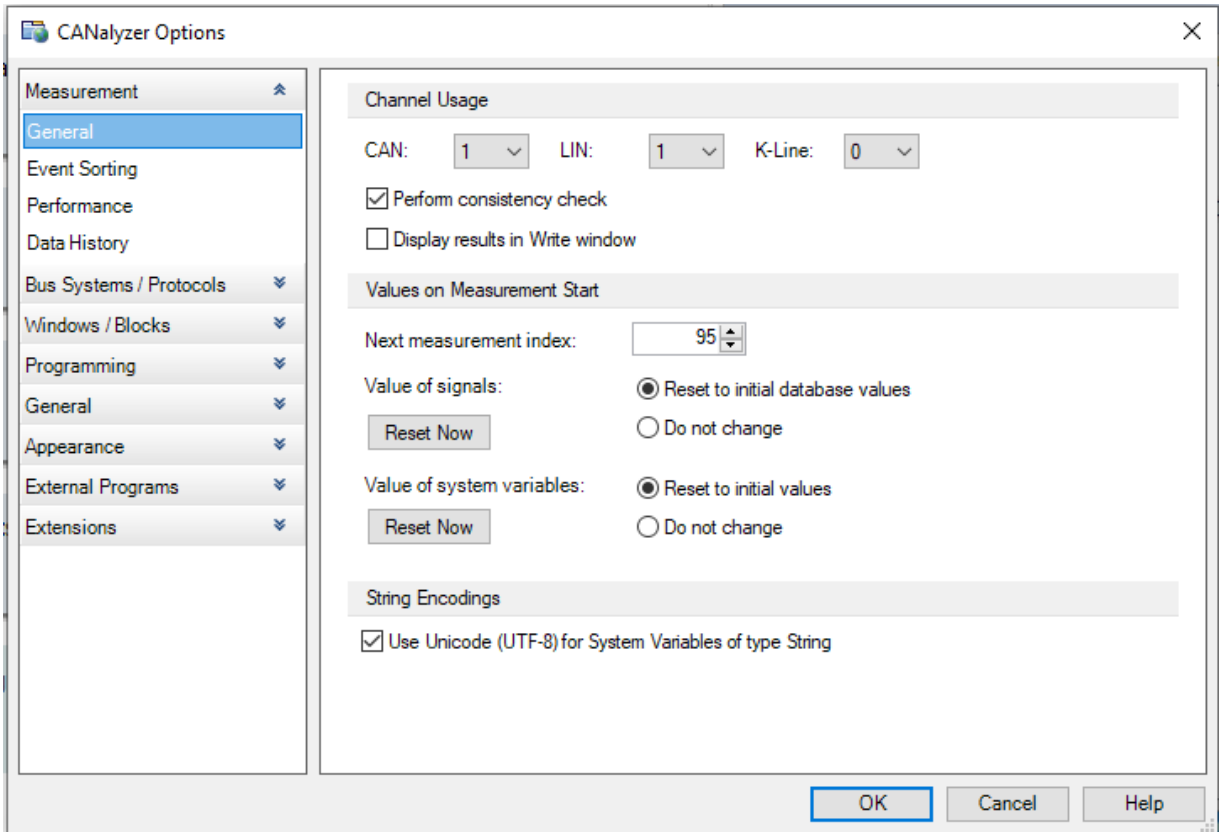


Figura 28. Uso de canales

Por otra parte, es conveniente configurar el **mapeado de canales** de CANalyzer, es decir, relacionar los canales de la CANcase, conectada al ordenador, a los canales CAN o LIN de nuestra configuración de CANalyzer. Con esa finalidad, vamos a "*Hardware*" y luego a "*Channel Mapping*". Aquí configuramos los canales correspondientes.

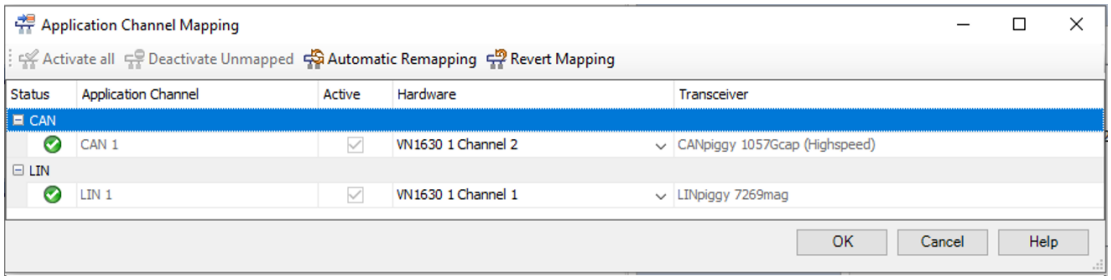


Figura 29. Mapeado de canales

4.8 Tools

Desde Tools, podremos acceder a **módulos integrados** de CANalyzer como CAPL browser, Panel Designer y CANdb++, los cuales explicaré más adelante en sus respectivos apartados.

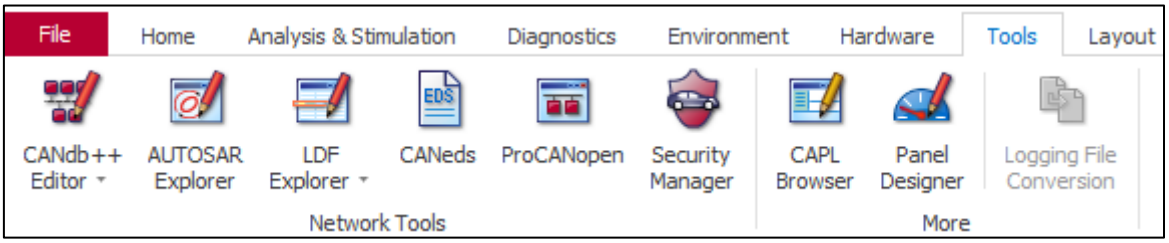


Figura 30. Opción Tools

5 Lenguaje y scripts CAPL

CAPL (*CAN Access Programming Language*) es el **lenguaje de programación interno de CANalyzer** y comparte varias similitudes con el lenguaje C++. CAPL es un **lenguaje dirigido por eventos**, es decir, cada script redactado en CAPL consta de funciones que reaccionan a **eventos externos**, tales como el envío de tramas, el inicio de una medición, la pulsación de un botón, etc. [3]

Usaremos el lenguaje CAPL para redactar scripts en el módulo integrado **CAPL browser** de CANalyzer. Para acceder a este módulo, hay que entrar en CANalyzer y hacer click en "*Tools*" y seleccionar "*CAPL browser*".

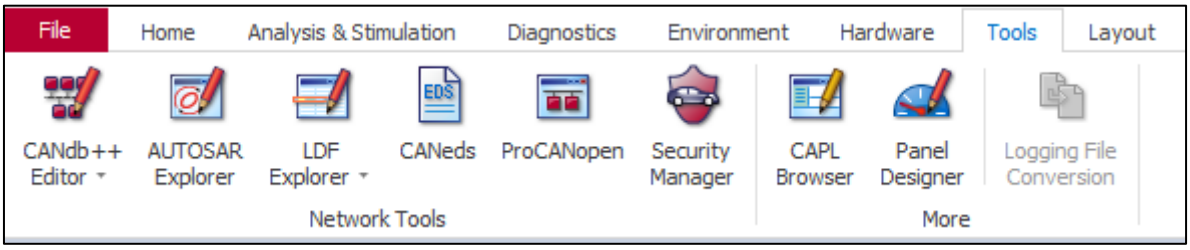


Figura 31. Opción Tools de CANalyzer.

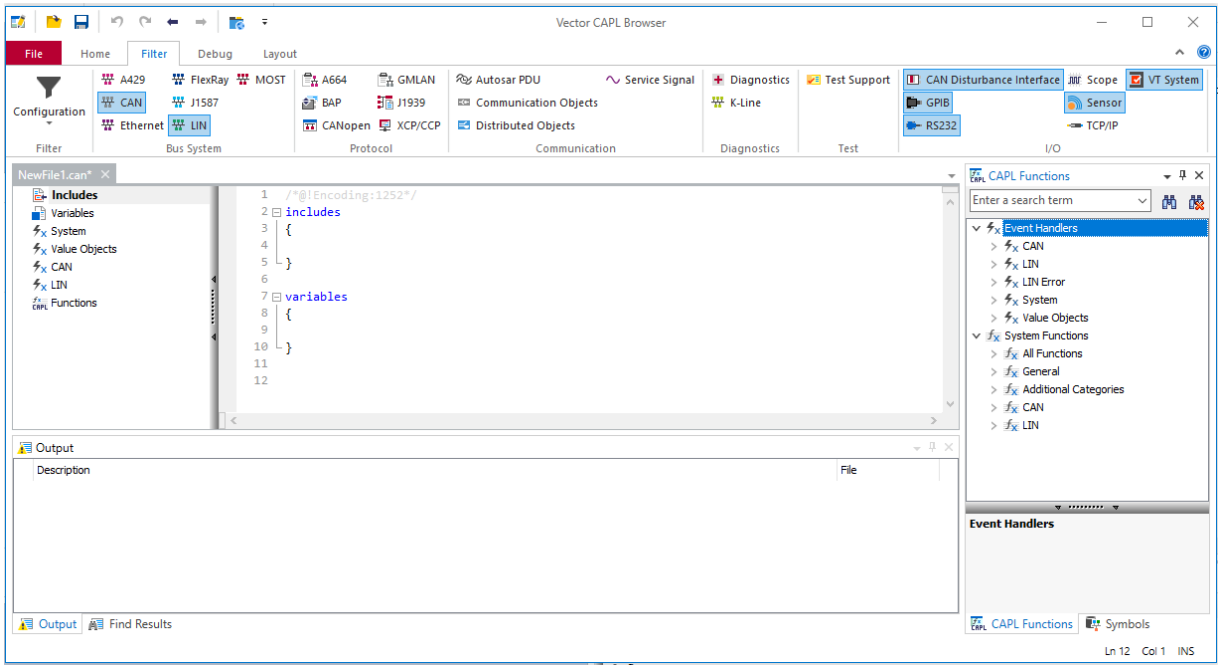


Figura 32. Ventana inicial de CAPL browser.

5.1 Comandos relevantes

`message CAN1.Message1 msgMessage1`. Nos permite llamar a la trama Message1 procedente de la mensajería asociada al canal CAN1 de CANalyzer y, basándonos en dicha trama, definir una trama interna msgMessage1 con la que podremos trabajar en los scripts CAPL. Este comando solamente funciona con mensajería de protocolo CAN.

`linframe Message1 msgMessage1`. Nos permite llamar a la trama Message1 procedente de la mensajería asociada al canal LIN1 de CANalyzer y, basándonos en dicha trama, definir una trama interna msgMessage1 con la que podremos trabajar en los scripts CAPL. Este comando solamente funciona con mensajería de protocolo LIN.

`Export void function1()`: Declaramos una función function1 que, al ser ejecutada, realiza acciones pero no devuelve ningún parámetro de salida.

`Write("text"):` Redacta la línea de texto "text" que figurará en la ventana "*Write*" de CANalyzer.

`Stop():` Detiene la medición de CANalyzer en plena ejecución.

`Includes`: Permite vincular el script actual a otros scripts redactados en CAPL o incluso otros documentos. Así, podemos usar variables y funciones definidas en otros scripts.

5.2 Event Handlers

Los Event Handlers son comandos que ejecutarán las líneas de código asociadas una vez tenga lugar un **evento** determinado.

`On Start.` Se ejecuta al iniciar una medición de CANalyzer.

`On Message msgTrama1.` Se ejecuta al recibir la trama *msgTrama1*.

`On timer timer1.` Se ejecuta al inicializar el temporizador *timer1*.

`On sysvar PC::variable1.` Se ejecuta cuando cambia el valor de la variable de sistema *variable1* del entorno *PC*.

5.3 Funciones relevantes

`sysGetVariableInt(sysvar::PC::Variable1).` Permite extraer un valor entero de la variable de sistema Variable1 declarada en el entorno PC.

`sysSetVariableInt(sysvar::PC::Variable1, 1).` Permite introducir un valor entero, 1, en la variable de sistema Variable1 declarada en el entorno PC. También sería posible introducir el valor de una variable de sistema diferente de la siguiente forma. `sysSetVariableInt(sysvar::PC::Variable1, @PC::Variable1).`

`sysSetVariableFloat(sysvar::PC:: Variable1, 1.0).` Permite introducir valores de tipo flotante en diferentes estructuras.

5.4 Temporizadores

A veces, en un script es necesario hacer que un comando se ejecute en un momento futuro, en vez del instante de ejecución. Con ese fin, utilizamos **temporizadores** para realizar acciones una vez finalizado el intervalo de tiempo deseado.

Los comandos relevantes para el uso de temporizadores son los siguientes:

`timer timer1.` Sirve para declarar el temporizador timer1, que cuenta en segundos.

`mstimer mstimer1.` Sirve para declarar el temporizador mstimer1, que cuenta en milisegundos.

`setTimer(timer1, t).` Nos permite inicializar el temporizador timer1 al cabo de t segundos.

`cancelTimer(timer1).` Detiene el temporizador timer1.

5.5 Nodo de programa

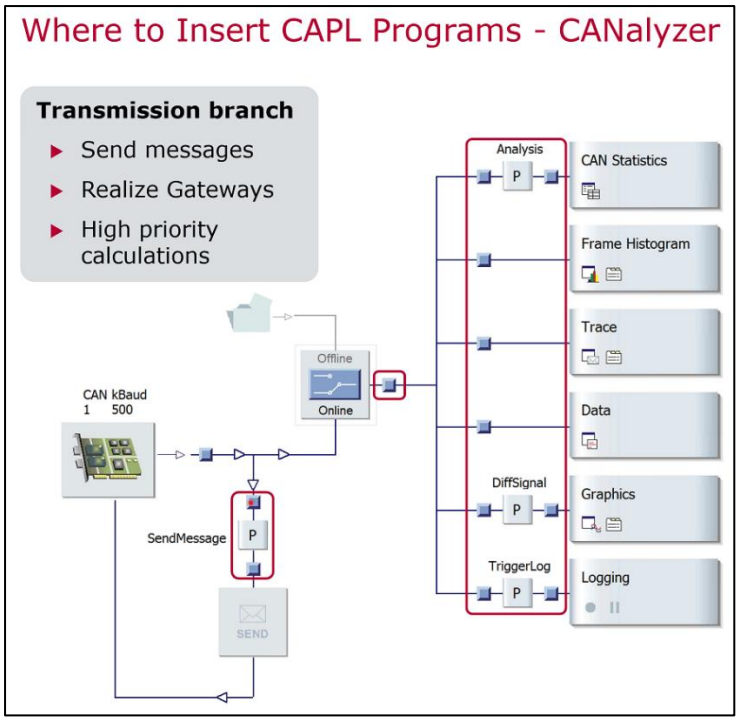


Figura 33. Nodo de programa en la ventana *Measuremente Setup* de CANalyzer. [4]

Es imperativo asociar el script CAPL relevante a un **nodo de programa** en la ventana *Measurement Setup* (**véase fig. 11**), estando el switch habilitado en modo online, para que nuestra configuración de CANalyzer pueda enviar y recibir tramas durante la medición, así como hacer que el script reconozca las bases de datos y entornos de variables de sistema asociados a la configuración. Para nuestro proyecto, el nodo de programa iría entre los dos *hotspots* abajo a la izquierda.

Además, para comprobar que **el script compila correctamente** desde el navegador de CAPL, hay que **abrir los scripts desde dichos nodos de programa**, no a través del navegador de CAPL.

6 Panel Designer

Panel Designer se trata de un módulo integrado de CANalyzer que nos permite diseñar interfaces gráficas que permiten al usuario tanto enviar demandas a ECUs del vehículo así como visualizar valores actuales de señales de los mismos. Para acceder a este módulo, hay que entrar en CANalyzer y hacer click en *Tools* y seleccionar *Panel Designer*.

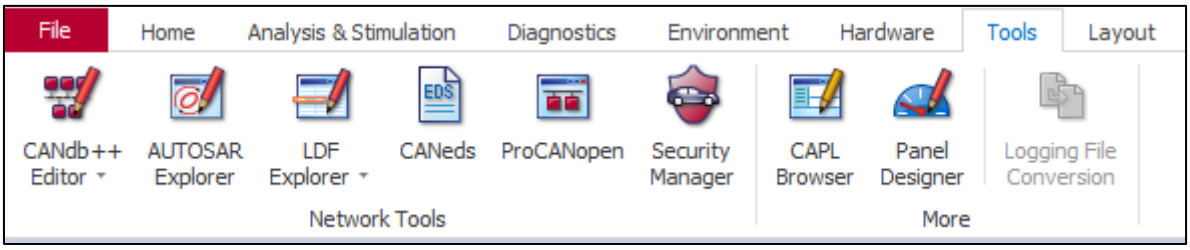


Figura 34. Opción Tools de CANalyzer.

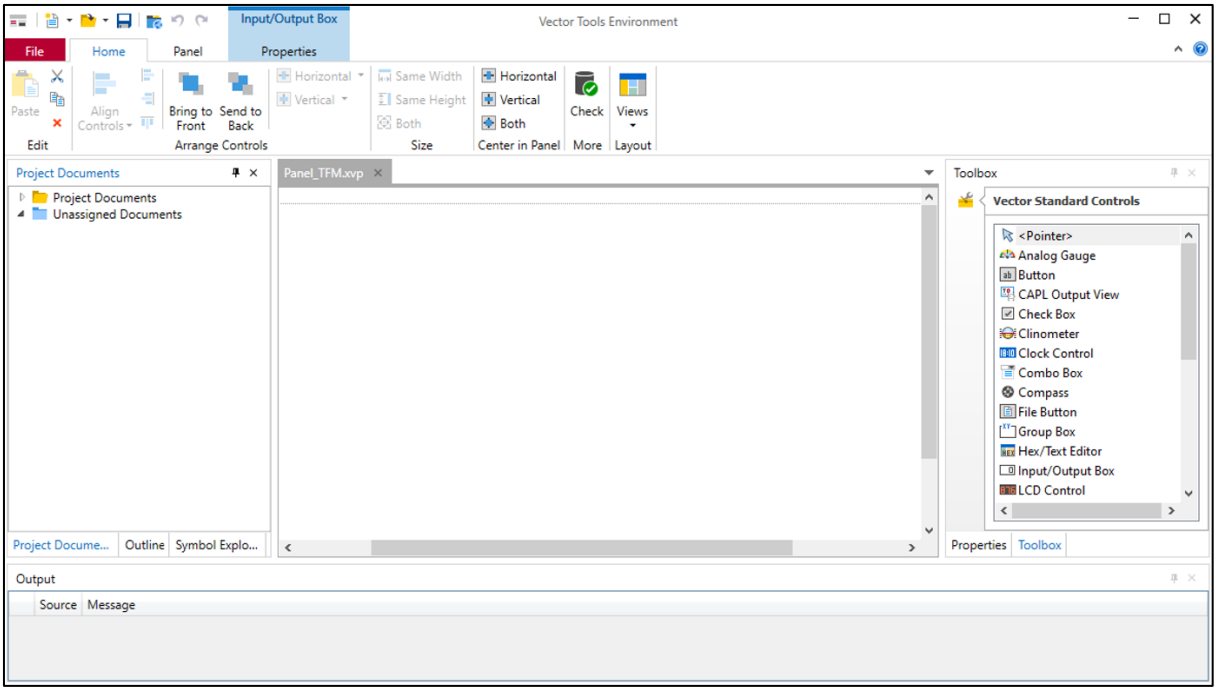


Figura 35. Ventana inicial de Panel Designer.

Desde la ventana *Outline*, podemos ver las variables de sistema y señales asociadas a cada elemento de la interfaz, así como datos pertinentes a cada una (el entorno generado para la variable de sistema y la trama, base de datos, nodo y red para las señales).

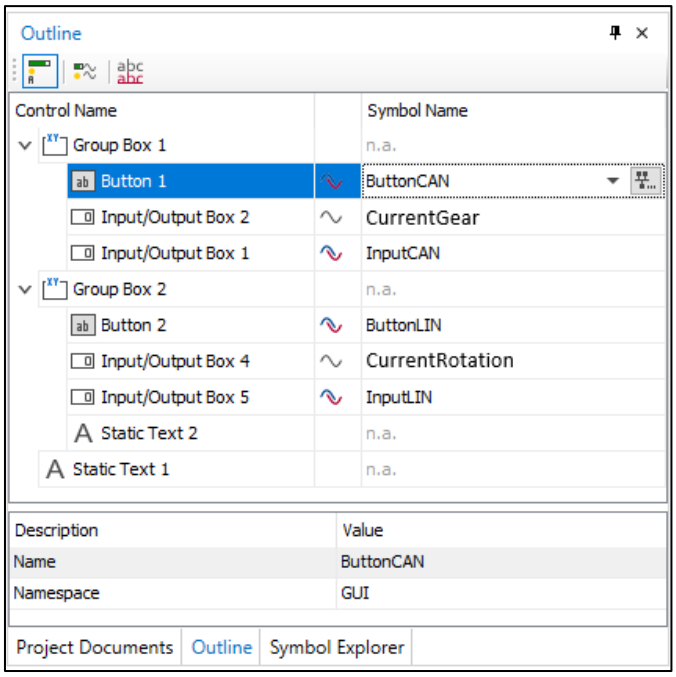


Figura 36. Ventana Outline de Panel Designer.

Desde la ventana *Symbol Explorer,* podemos seleccionar señales vinculadas a las bases de datos establecidas, así como variables de sistema de entornos generados, y asociarlas a determinados elementos de la ventana Toolbox.

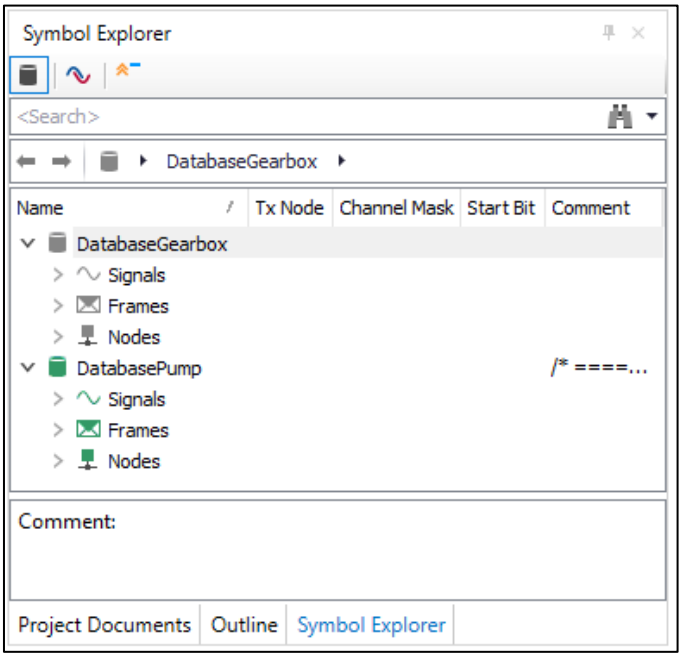


Figura 37. Ventana Symbol Explorer para señales y tramas.

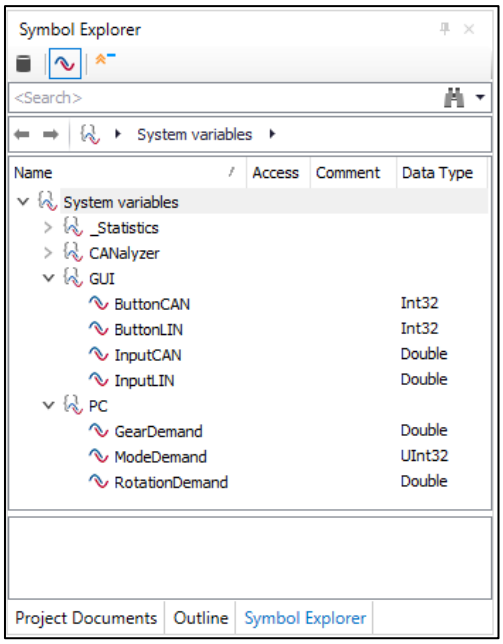


Figura 38. Ventana Symbol Explorer para variables de sistema.

Yendo a la ventana *Toolbox*, tenemos varios elementos a nuestra disposición para configurar la interfaz.

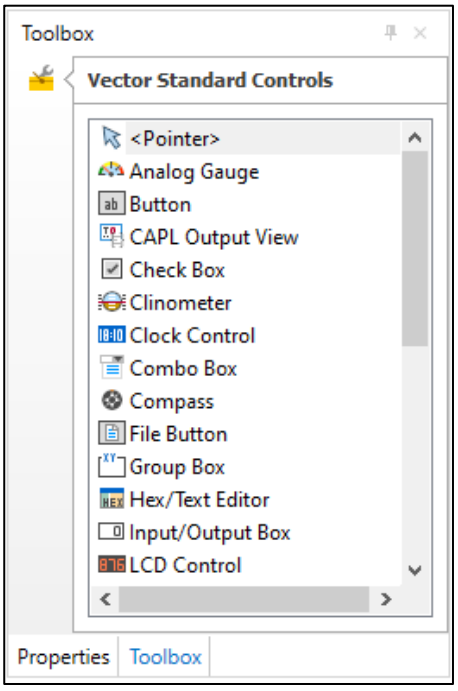


Figura 39. Toolbox de Panel Designer.

Para nuestro proyecto, los elementos más relevantes serán los siguientes:

Cajas Input/Output: Se tratan de cajetines que pueden actuar como **elementos de control** (donde el usuario puede introducir datos para definir una variable de sistema) o como **elementos de display** (solamente para visualización, nos devuelve el valor actual de una señal o variable de sistema).

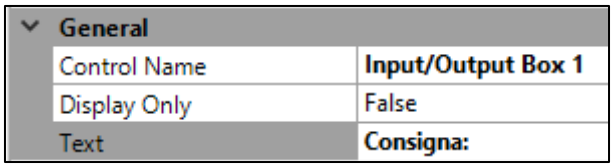


Figura 40. Parámetro Display Only de un cajetín Input/Output.

Con el parámetro *Display Only*, podemos configurar el tipo de caja.

- Si es asignada como "True", la caja es un elemento de display, sólo se podrá visualizar el valor actual de la señal o variable de sistema asociada.

- Si es asignada como "False", la caja es un elemento de control, se podrá asignar el valor deseado a la variable de sistema asociada.

Botones: Se tratan de elementos de control que constan de una **variable booleana** interna, es decir, una variable que solo puede adoptar **dos valores, 0 o 1**. El valor 1 está asociado al evento de que el **botón esté presionado**, y el valor 0, asociado al evento de que el **botón sea liberado**. Se recomienda asociar variables de sistema a botones y usar el event handler `on sysvar` con los mismos, de manera que, cuando el botón sea presionado, se ejecutarán las líneas de código correspondientes.

Figura 41. Elemento "Button" de Panel Designer.

Yendo a la ventana *Properties*, podemos visualizar las propiedades y parámetros de cada elemento de la interfaz.

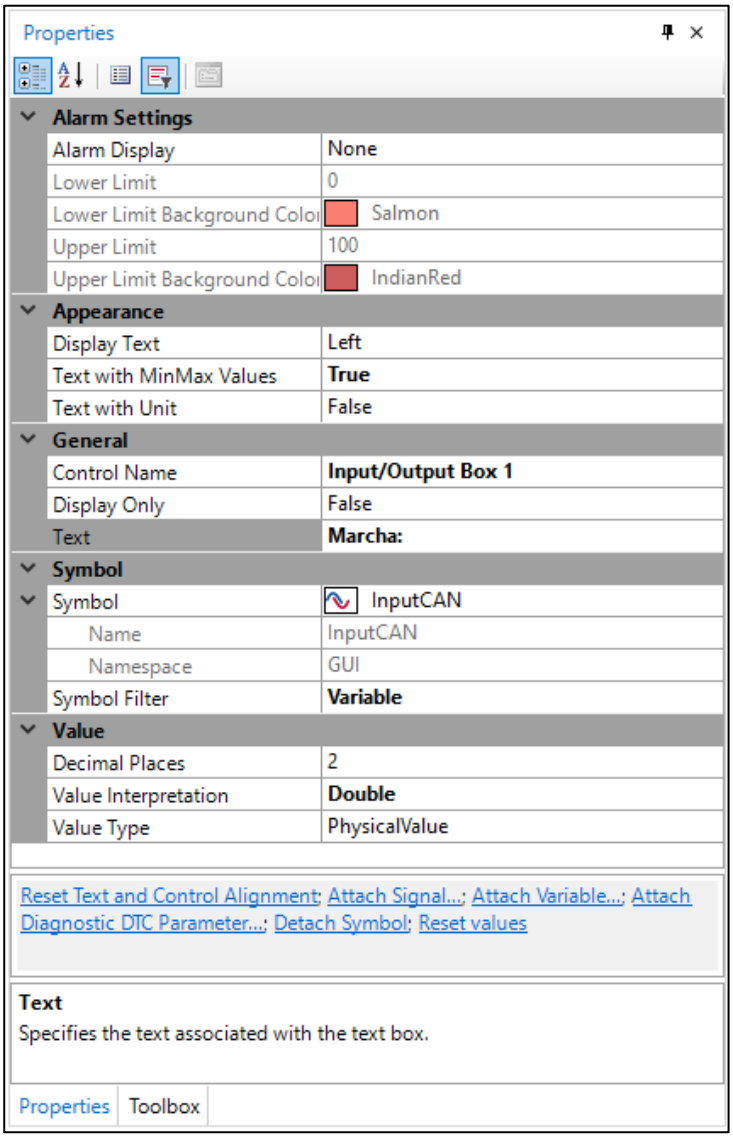


Figura 42. Ventana Properties.

7 CANdb+

Es posible crear y editar nuestras propias bases de datos mediante el módulo integrado **CANdb+** de CANalyzer. Para acceder a este módulo, hay que entrar en CANalyzer y hacer click en *Tools* y seleccionar *Panel Designer* **(véase fig. 27).**

7.1 Creación de bases de datos, tramas y señales

Desde el programa CANdb++ 3.1, se crea una base de datos nueva, pulsando "*File*" y luego "*Create database*". Escojo el formato CANTemplate.

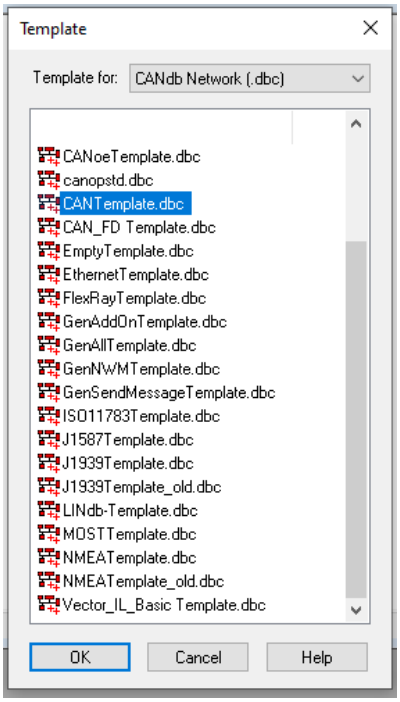


Figura 43. Formatos de bases de datos.

Después, hay que crear nodos, tramas y señales. Basta con hacer click derecho en "*Network Nodes*", "*Messages*" y "*Signals*" respectivamente y pulsar "*New…*"

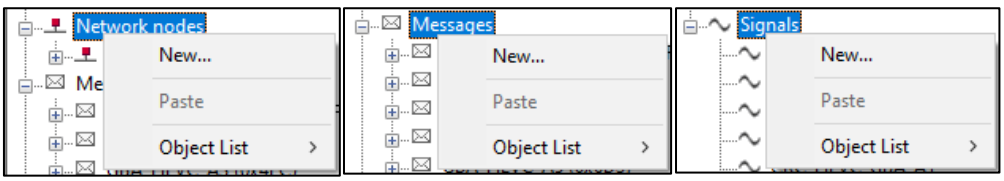


Figura 44. Creación de nodos, tramas y señales.

Posteriormente, asociamos señales a las tramas.

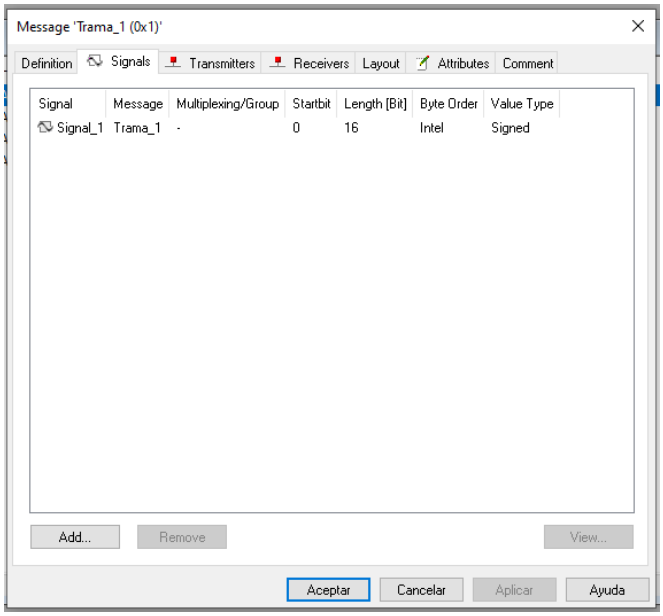


Figura 45. Diseño de tramas.

Luego, asociar las tramas a los nodos, haciendo click derecho y luego eligiendo *Edit Node*.

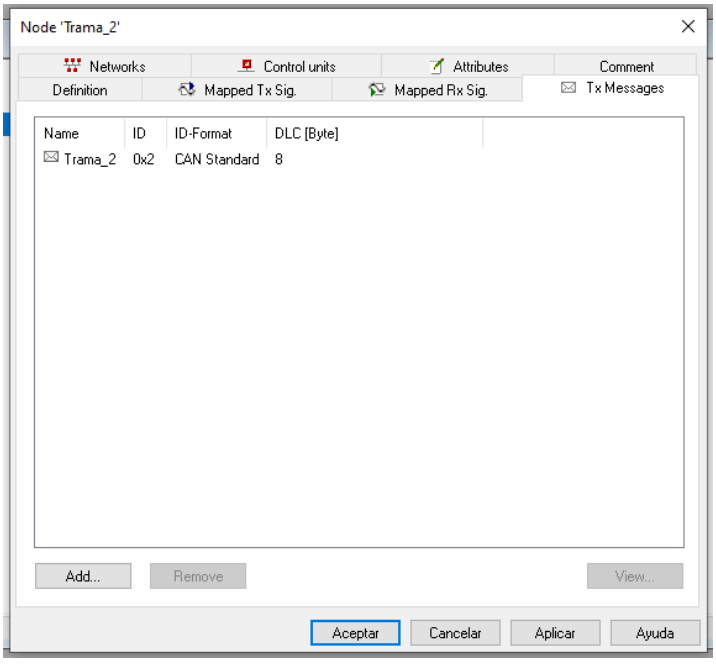


Figura 46. Diseño de nodos.

Así, ya tenemos la señal de dicha trama mapeada como señal TX del nodo automáticamente.

7.2 Edición de tramas

Distinguimos varias ventanas. Nos fijamos en la trama **MessageRXCAN** como ejemplo. Para empezar, en la ventana "*Definition*", establecemos el nombre de la trama, el número identificativo de la trama en sistema hexadecimal y el **DLC (Data-Length Code),** un parámetro que asigna el espacio máximo que pueden ocupar las señales dentro de la trama en bytes.

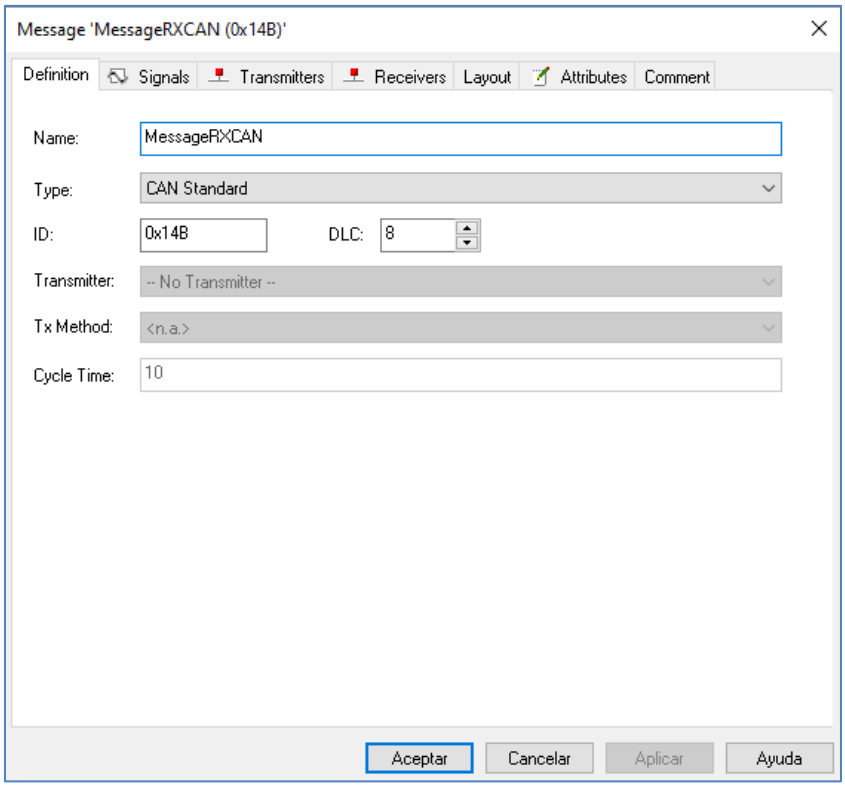


Figura 47. Ventana Definition.

Luego, tenemos la ventana Signals, donde asignamos las señales correspondientes a esta trama.

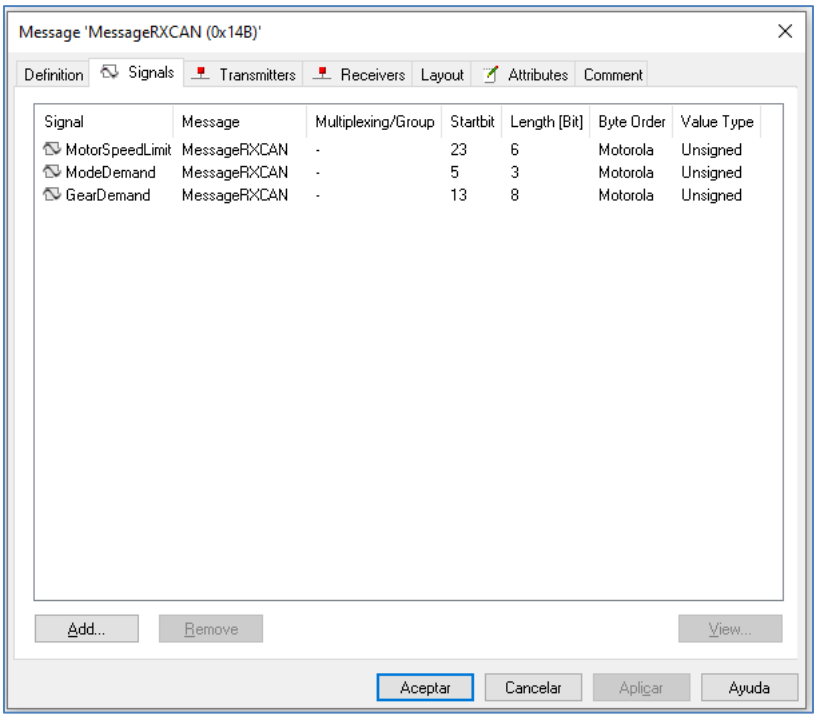


Figura 48. Ventana Signals.

A continuación, tenemos las ventanas "*Transmitters*" y "*Receivers*", donde asignamos los nodos correspondientes a las ECUs que van a transmitir o recibir, respectivamente, la trama. Como la trama MessageRXCAN será recibida por la ECU de la caja, asociamos el nodo "**ECU**" en "*Receivers*".

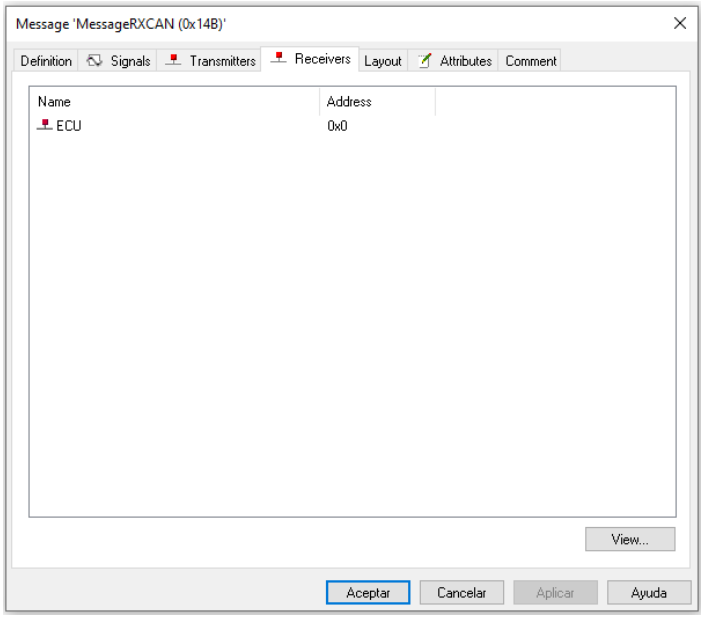


Figura 49. Ventana Receivers.

Después, tenemos la ventana "*Layout*", que nos muestra un esquema del espacio total en la trama, en filas de 8 bits enumeradas de arriba abajo y de derecha a izquierda (de 0 a 7, de 8 a 15, de 16 a 23, etc) y los bits que ocupa cada señal en dicha trama.

Hay que tener en cuenta, si una señal empieza por una fila de bits determinada y se extiende más allá de dicha fila, en vez de continuar por la fila de abajo, **se moverá a la fila de arriba**. Por ejemplo, si una señal empieza en la tercera fila, de 16 a 23, y se extiende más allá del bit 23, en vez de continuar por el bit 24 en la fila de abajo, va a continuar por el bit 8 en la fila de arriba.

En conclusión, **el bit 0 va después del bit 15, el bit 8 va después del bit 23**, etc.

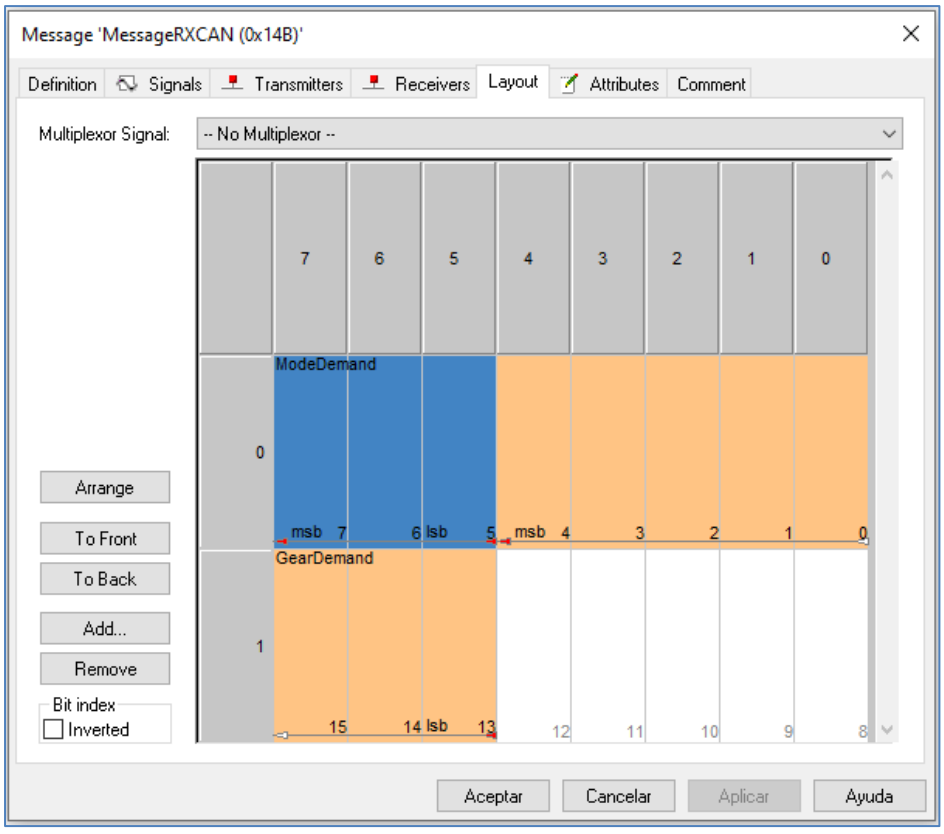


Figura 50. Ventana Properties.

7.3 Edición de señales

Desde la ventana "*Definition*", podemos ver la trama en la que se encuentra la señal y el nombre de la misma así como editar el bit de inicio de la señal dentro de la trama y el tipo de señal.

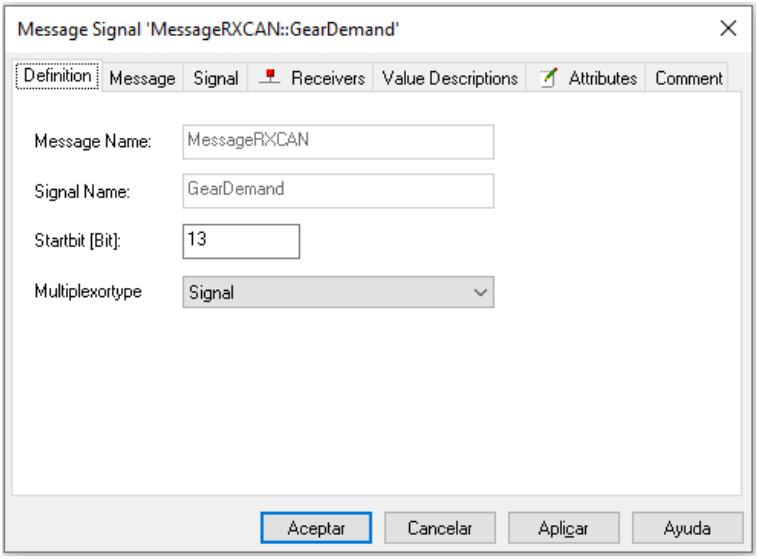


Figura 51. Ventana Definition.

En la ventana "Message", vemos la misma información que en la ventana Message de la trama. No es posible editar ningún de estos datos desde aquí, solamente se puede desde la trama.

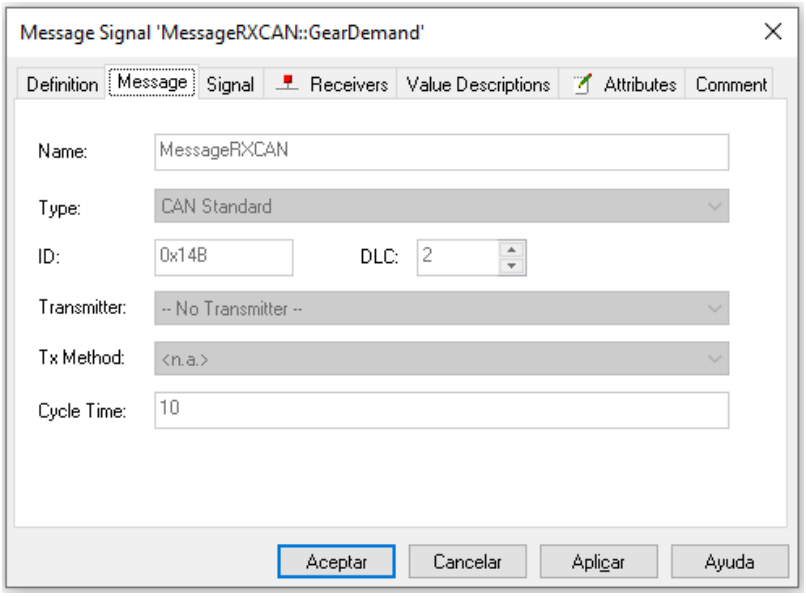


Figura 52. Ventana Message.

Desde la ventana "*Signal*", podemos editar varios parámetros:

- El nombre de la señal.
- La **memoria en bits** que dicha señal ocupará dentro de la trama.
- El **tipo de valor** (*Signed, Unsigned, IEEE Float e IEEE Double*).
- La unidad empleada y valores mínimo y máximo de la señal.
- El **factor,** la resolución en la que dividimos los valores de la señal.
- El **offset**, el valor inicial retardado o adelantado respecto a 0, el valor inicial por defecto, siempre que no sea inferior al valor mínimo o superior al máximo fijados.
- Una **tabla de valores**, con función puramente descriptiva, que indicará el significado asociado a cada posible valor de la señal. Dichos valores figurarán en la ventana "*Value Descriptions*".

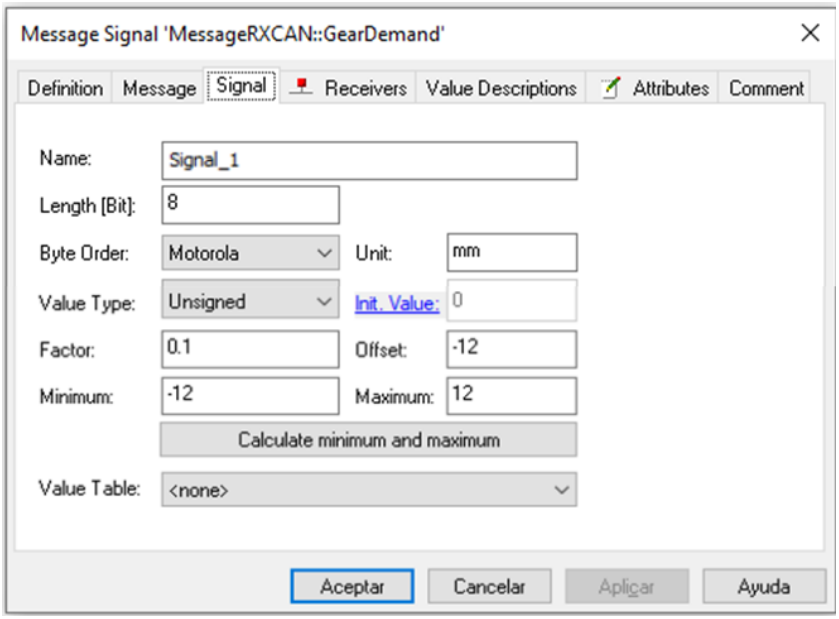


Figura 53. Ventana Signal.

Pulsando el botón "*Calculate minimum and maximum*", podemos hallar el **valor máximo que podría almacenar la señal**, teniendo en cuenta la memoria en bits, el valor mínimo y el factor. Por ejemplo, para la señal *Signal_1*, estando comprendida entre -12 mm y 12 mm y teniendo un factor de 0.1 mm, la señal contendrá **241 posibles valores, incluyendo el valor inicial** $[\frac{12-(-12)}{0.1} + 1 = 241]$. Sin embargo, teniendo en cuenta que la señal mide 8 bits en memoria, podría almacenar hasta **256 valores** $[2^8 = 256]$. Así, quedan **15 valores vacíos**. Por lo tanto, pulsando "*Calculate minimum and maximum*", tendremos un valor máximo de **13.5 mm** $[15 * 0.1 + 12 = 13.5\ mm]$.

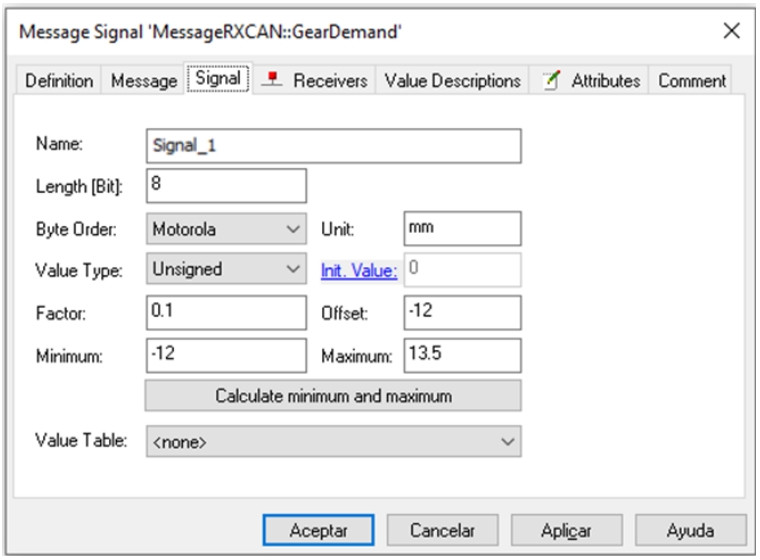


Figura 54. Efecto de Calculate minimum and maximum.

Esta función es útil para determinar si los bits asignados a la señal son suficientes para englobar todos los valores posibles, o para ampliar el rango de valores si queda espacio libre en la señal

APARTADO 3: DISEÑO

1 Caso de estudio: Pilotado de ECUs mediante interfaz de usuario

1.1 Propuesta de interfaz

Para el diseño de la interfaz de usuario que nos permitirá satisfacer la demanda del cliente, se ha empleado el módulo integrado **Panel Designer** de CANalyzer.

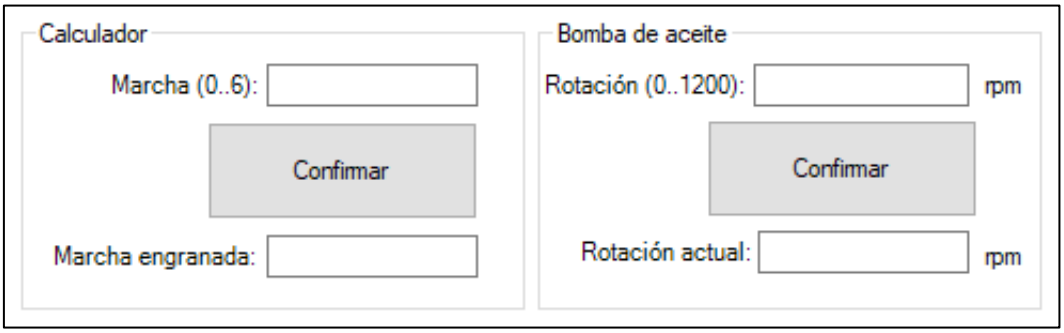


Figura 55. Diseño de interfaz.

La interfaz dispone de dos **groupboxes**, una para cada componente del vehículo con el que trabajaremos, el calculador y la bomba de aceite. Cada groupbox dispone de los siguientes elementos:

- Una caja "**Consigna**", un elemento de control donde el usuario introduce el valor deseado del parámetro. Para nuestra interfaz, tendremos la relación de marcha para el calculador y el régimen de rotación, de 0 a 1200 rpm, para la bomba de aceite.

- Un botón "**Confirmar**", el cual, al ser pulsado, ejecuta un script que envía la consigna introducida por el usuario al elemento deseado del vehículo.

- Una caja "**Salida**", un elemento de display que nos devuelve el valor real del parámetro después de la ejecución del script. En nuestro caso, la marcha engranada actualmente y velocidad de rotación actual. En condiciones ideales, el valor de salida ha de coincidir con el valor de entrada. A estas salidas fijaremos las señales CurrentGear y CurrentRotation para el calculador y la bomba respectivamente.

1.2 Entornos y variables de sistema

Por una parte, cuando diseño la interfaz en Panel Designer y, por ejemplo, asigno la señal deseada a una caja "Comando", el parámetro "Display Only" está fijado a True, no se puede modificar, de forma que esta caja no funciona como elemento de Control, que es lo que nos interesa tener, sino como elemento de Display solamente.

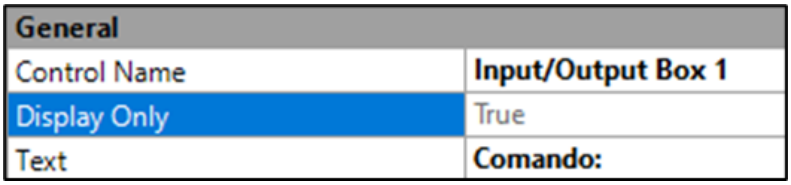


Figura 56. Propiedades de Input/Output Box.

Por otra parte, no es posible asociar señales a botones mediante Panel Designer.

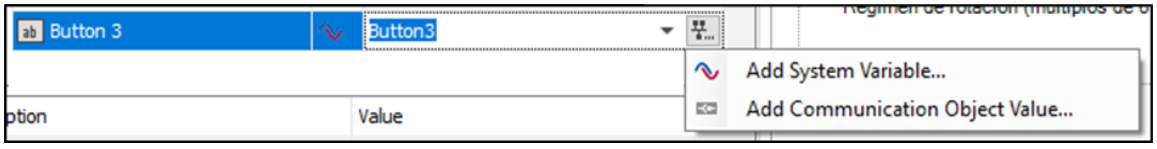


Figura 57. Variable de sistema asociada al botón.

Para poder solventar estos problemas, en vez de asignar señales a los elementos de panel directamente, tenemos que usar **variables de sistema**, que irán agrupadas en dos entornos generados, PC y GUI.

- El **entorno GUI (Graphic User Interface).** Este entorno incluye las **variables de entrada,** asociadas a los elementos de consigna de la interfaz, que recogerán los valores introducidos por el usuario (variables **InputCAN e InputLIN**). Después, usando scripts, transferiremos los valores de dichas variables a las variables correspondientes del entorno PC. En este entorno, también definiremos las **variables de pulsación** asociadas a los botones (variables **ButtonCAN y ButtonLIN**). Estas variables serán booleanas, es decir, sólo adoptan 0 o 1 como valores, cuando el botón está libre o es presionado .

- El **entorno PC.** Mediante scripts, este entorno reciben los valores correspondientes del entorno GUI y, posteriormente, transferiremos dichos valores a las señales relevantes de nuestras bases de datos. Cada una de estas variables se llama de forma idéntica a la señal correspondiente (**GearDemand, ModeDemand y RotationDemand**).

Cabe destacar, los **parámetros** de las variables de entrada, de las variables del entorno PC y de las señales correspondientes han de **coincidir entre sí**. Así, la variable **GearDemand** recogerá el valor de InputCAN y lo transferirá a la señal GearDemand del dbc, mientras que la variable **RotationDemand** recogerá el valor de InputLIN y lo transferirá a la señal RotationDemand del ldf. Por otra parte, definiremos la variable **ModeDemand** en el script posteriormente y luego la enviaremos al calculador.

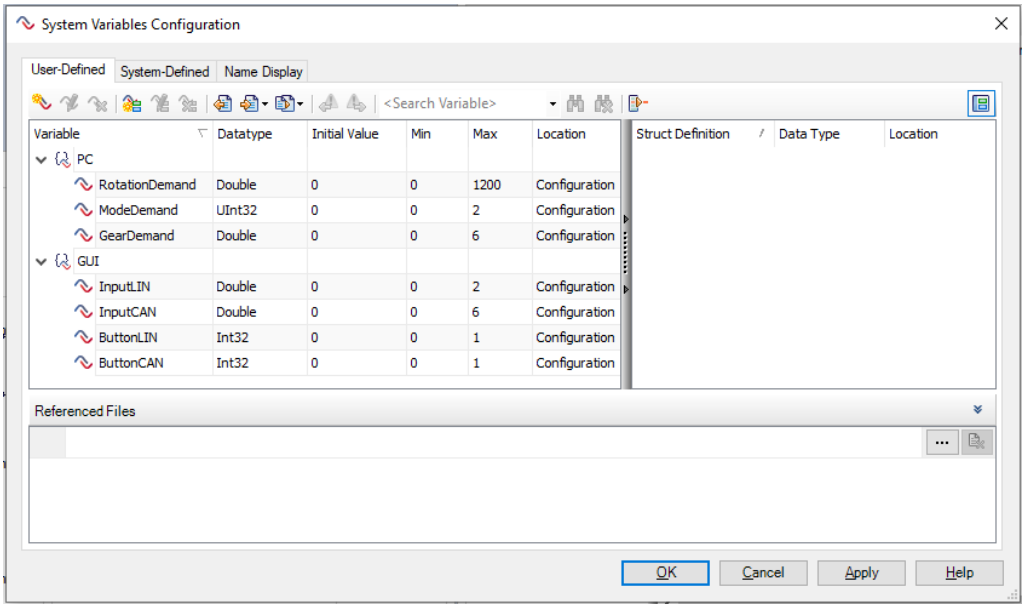


Figura 58. Entornos generados

A la hora de fijar valores iniciales, máximos, mínimos y unitarios en cada variable, he usado CANdb++ y ldf explorer para abrir los archivos dbc y ldf respectivamente y me he fijado en los parámetros de la señal correspondiente a cada variable, de manera que la información sea coherente.

1.3 Edición de scripts

Para la programación de scripts asociados a la interfaz de usuario, emplearé el módulo integrado **CAPL browser** de CANalyzer.

1.3.1 Definición de tramas y señales

En el script **CAN_out.cin**, llamo las tramas MessageRXCAN y MessageRXLIN, del dbc y del ldf respectivamente, y las defino internamente.

```
Variables
{
  message CAN1.MessageRXCAN msgMessageRXCAN;
  linframe MessageRXLIN msgMessageRXLIN;
```

Figura 59. Identificación de tramas.

Después, defino dos funciones, *WriteMessageRXCAN* y *WriteMessageRXLIN*, las cuales, cuando sean ejecutadas posteriormente, **transferirán los valores de las variables de sistema del entorno PC a las señales correspondientes**.

```
export void WriteMessageRXCAN()
{
  msgMessageRXCAN.ModeDemand.phys = sysGetVariableInt(sysvar::PC::ModeDemand);
  msgMessageRXCAN.GearDemand.phys = sysGetVariableInt(sysvar::PC::GearDemand);
}

export void WriteMessageRXLIN()
{
msgMessageRXLIN.RotationDemand.phys=sysGetVariableInt(sysvar::PC::RotationDemand);
}
```

Figura 60. Transferencia de valores de variables a señales.

1.3.2 Definición de temporizadores

Después de definir las tramas, tenemos que **enviarlas al componente deseado del vehículo**. Nos interesa que ambos eventos tengan lugar en instantes futuros consecutivamente, en vez del propio instante de medición en CANalyzer simultáneamente. Con ese propósito, tenemos que utilizar **temporizadores**.

Así, en el script **CAN_out.cin**, definimos dos temporizadores, **tmrMessageRXCAN y tmrMessageRXLIN**, que enviarán cada trama RX periódicamente.

```
  //Temporizadores para enviar mensajes cíclicos
  msTimer tmrMessageRXCAN;
  msTimer tmrMessageRXLIN;
}
```

Figura 61. Definición de temporizador.

```
on timer tmrMessageRXCAN
{
  output(msgMessageRXCAN);
  setTimer(tmrMessageRXCAN, 10 + random(2));
}

on timer tmrMessageRXLIN
{
  output(msgMessageRXLIN);
  setTimer(tmrMessageRXLIN, 10 + random(2));
}
```

Figura 62. Envío periódico de tramas.

Después, en el script **Output.can**, defino dos funciones:

- **PreConditions():** una vez ejecutada, inicializará los temporizadores.
- **PostConditions():** una vez ejecutada, detendrá los temporizadores.

```
export void Preconditions()
{
  setTimer(tmrMessageRXCAN, 10); //10 ms
  setTimer(tmrMessageRXLIN, 5); //5 ms
}

export void Postconditions()
{
  cancelTimer(tmrMessageRXCAN);
  cancelTimer(tmrMessageRXLIN);
}
```

Figura 63. Inicialización y cancelación de temporizadores

En el script **Input.can,** definimos un temporizador auxiliar, **tmrButton_Aux**, que ejecutará PostConditions(), deteniendo así el envío de tramas. El tiempo de inicialización de este temporizador tendrá que ser superior al de los temporizadores que envían tramas, de forma que la cancelación de temporizadores tenga lugar después del envío de tramas.

```
variables
{
  msTimer tmrButton_Aux;
}
```

Figura 64. Definición de temporizador auxiliar.

```
on timer tmrButton_Aux
{
      PostConditions();
}
```

Figura 65. Cancelación de temporizadores.

1.3.3 Pulsación de botones

El evento que ejecutará el script **Input.can** será cambiar el valor de uno de los botones del panel, "`on sysvar GUI::ButtonCAN`" u "`on sysvar GUI::ButtonLIN`". Con esta primera línea de código, **cualquier cambio en el botón podría ejecutar el código**, ya sea presionar el botón o liberarlo, por lo que es necesario especificar luego que sólo nos interesa que el código sea ejecutado cuando el botón sea pulsado.

```
on sysvar GUI::ButtonCAN
{
  if (@GUI::ButtonCAN==1)
    {
      PressButtonCAN();
    }
}
```

```
on sysvar GUI::ButtonLIN
{
  if (@GUI::ButtonLIN==1)
    {
      PressButtonLIN();
    }
}
```

Figura 66. Pulsación de botones.

Si el valor de la variable de sistema asociada al botón pasa de 0 a 1, "`if (@GUI::ButtonCAN==1)`" o "`if(@GUI::ButtonLIN==1)`", (es decir, cuando el botón es presionado), entonces el código asignado es ejecutado. Como se puede observar, el código no se ejecuta cuando el valor de la variable cambia de 1 a 0 (cuando el botón es liberado), de esa manera evitando redundancias y errores.

1.3.4 Pilotaje del componente de vehículo

Al pulsar cada botón, se ejecutará una función (**PressButtonCAN() y PressButtonLIN()** respectivamente), la cual, si el valor introducido en la interfaz es admisible, transferirá los valores de la variables InputCAN e InputLIN del entorno GUI a las variables **GearDemand y RotationDemand** del entorno PC.

Luego, mediante las funciones writeMessageRXCAN() y writeMessageRXLIN() definidas anteriormente, transferimos los valores de las variables GearDemand y RotationDemand a las señales homónimas respectivas.

Finalmente, ejecutamos Preconditions(), **enviando las tramas periódicamente** y, después, inicializamos el temporizador auxiliar que ejecutará PostConditions(), **deteniendo el envío de tramas.**

```
Void PressButtonCAN()
{
  If(@GUI::InputCAN<0 || @GUI::InputCAN>6
    {
      Write("No ha introducido una marcha correcta")
    }
  Else
    {
      sysSetVariableInt(sysvar::PC::ModeDemand, 1)
      sysSetVariableInt(sysvar::PC::GearDemand, @GUI::InputCAN)

      writeMessageRXCAN();
      Preconditions();
      setTimer(tmrButton_Aux, 300);
      write("Ha introducido una marcha correcta")
    }
}
```

Figura 67. Definición de la trama MessageRXCAN.

```
Void PressButtonLIN()
{
  If(@GUI::InputLIN<0 || @GUI::InputLIN>1200
    {
      Write("No ha introducido un valor correcto")
    }
  Else
    {
      sysSetVariableInt(sysvar::PC::RotationDemand, @GUI::InputLIN)

      writeMessageRXLIN();
      Preconditions();
      setTimer(tmrButton_Aux, 300);
      write("Ha introducido un valor correcto")
    }
}
```

Figura 68. Definición de la trama MessageRXLIN.

2 Caso de estudio: Diagnóstico de tráfico de datos

Para poder realizar la prueba en vehículo posteriormente, primero tenemos que **compilar el modelo Simulink y cargarlo en la MUXlab**.

2.1 Compilación del modelo Simulink

Partimos de un modelo simulink, aportado por el cliente, que manda consultas a ECUs y realiza envíos al banco de ensayos. Abrimos el modelo en la extensión Simulink de MATlab y **compilamos el modelo**, eligiendo la aplicación "*Simulink Coder*" y luego pulsando el botón "*Build*".

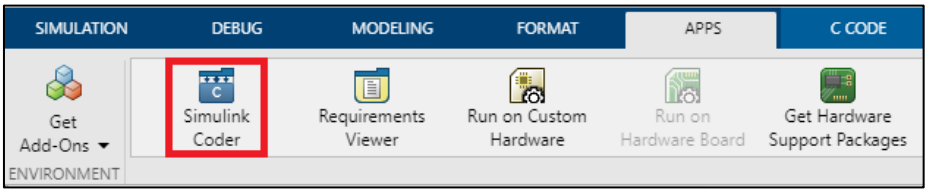


Figura 69. Aplicación Simulink Coder.

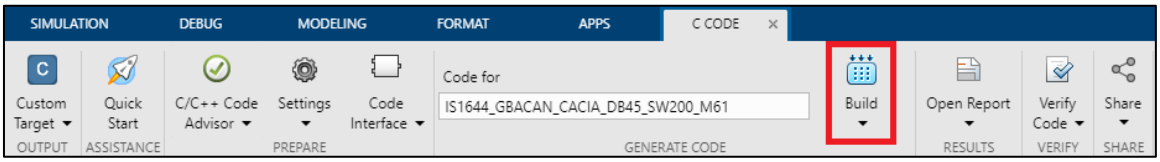


Figura 70. Compilación del modelo Simulink.

Después de compilar el modelo, se generará un **fichero .mot** que cargaremos en la MUXlab posteriormente. Dicho fichero se genera en la carpeta *nsi_rtw,* dentro de la subcarpeta *bin.*

Figura 71. Ficheros generados tras la compilación.

2.2 Carga del modelo en la MUXlab con USB firmware updater

Para poder cargar el modelo Simulink en la MUXlab, tenemos que conectar nuestro ordenador a la MUXlab mediante un cable USB tipo B, luego conectar la MUXlab a una fuente de tensión mediante dos bananas, roja y negra, y finalmente arrancar el programa **USB firmware updater**. Pulsamos "*Browse"* para seleccionar el fichero .mot generado anteriormente y luego pulsamos "*Flash program"* para cargar el modelo en la MUXlab. Una vez terminado el proceso, en la ventana tiene que poner *"successfully flashed*" para confirmar que el modelo se ha cargado correctamente.

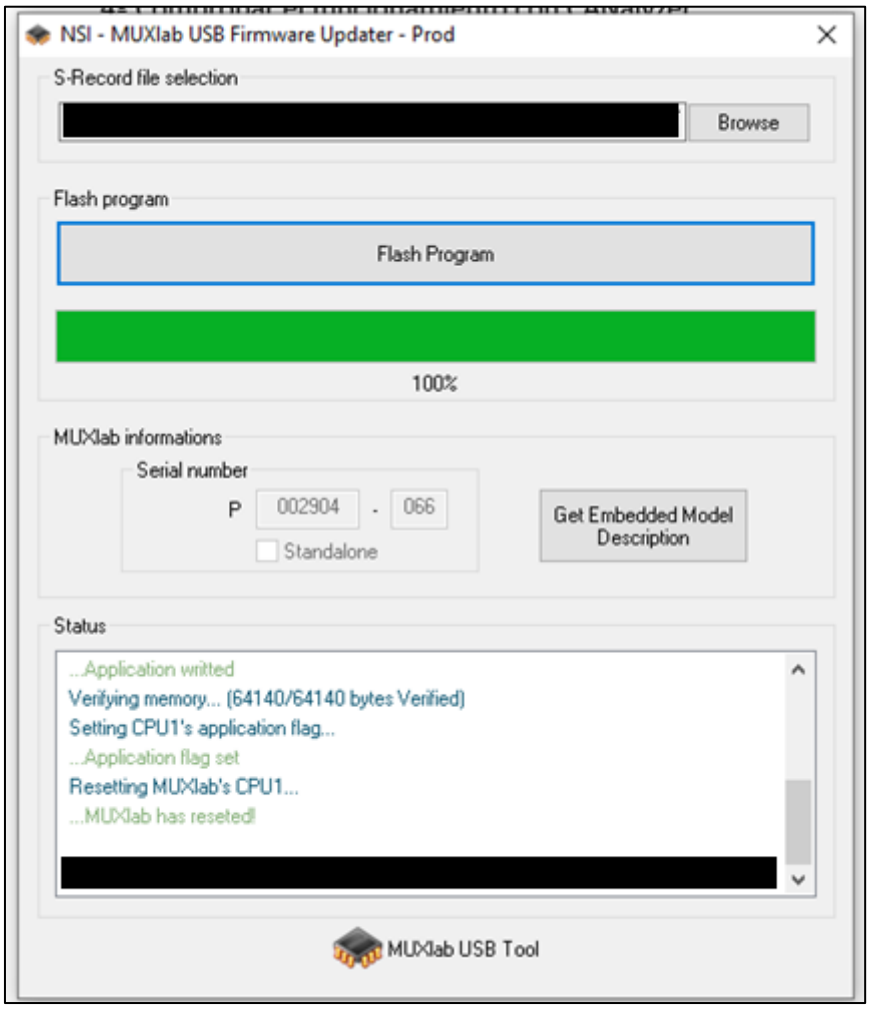


Figura 72. Flasheado del modelo Simulink.

Pulsamos el botón "*Get Embedded Model Description"* para obtener información del modelo flasheado y así verificar que hemos cargado el modelo correcto en la MUXlab.

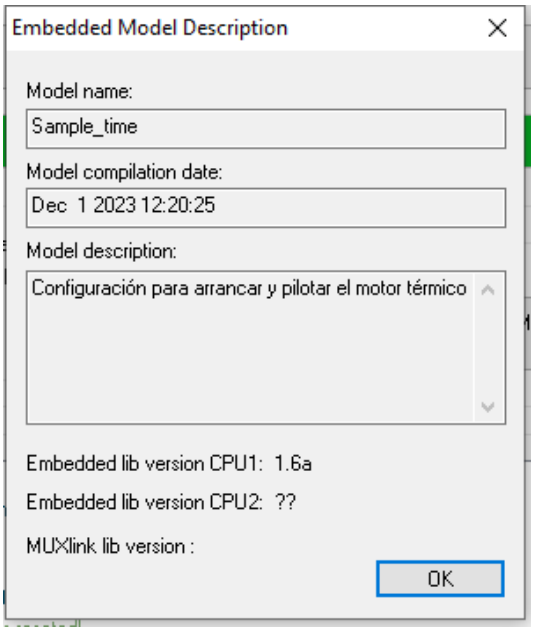


Figura 73. Descripción del modelo.

En la MUXlab, estarán encendidas las luces correspondientes a los canales CAN empleados por el modelo Simulink, en verde, y parpadeará a 1 Hz la luz de Status en rojo.

Figura 74. Descripción del modelo.

Así, ya tenemos preparada la MUXlab para la posterior conexión y prueba en vehículo.

APARTADO 4: PRUEBAS

1. Pilotado de ECUS mediante interfaz de usuario

He realizado varias pruebas en un laboratorio con el hardware y software necesarios, habiendo realizado el cableado necesario (**veáse fig. 55 y 56**). A continuación, detallo mis conclusiones.

Empezamos tratando con el **calculador**. Primero, hice una prueba introduciendo marchas correctas. A los 3 segundos engrané la marcha 2, a los 6 segundos la marcha 4 y a los 9 segundos la marcha 6. En cada caso, el calculador ha reaccionado correctamente y nos ha devuelto la señal TX CurrentGear con el valor correspondiente.

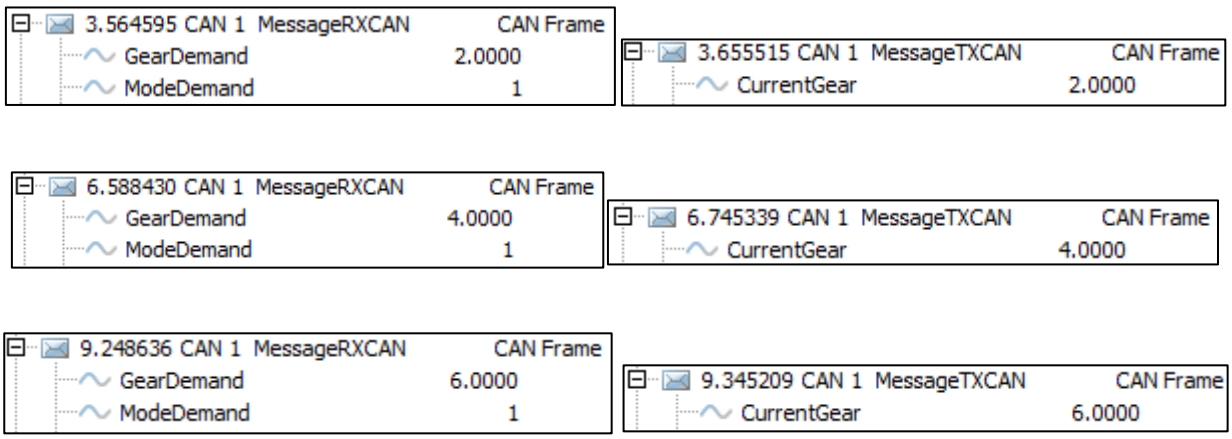


Figura 75. Verificación del pilotado del calculador.

Podemos verificar los resultados en la ventana Trace. Podemos ver que enviamos los valores en la trama RX y el calculador nos devuelve estos valores en la trama TX. Después, hice una prueba introduciendo una consigna fuera de rango, para comprobar que la configuración no ejecuta los scripts con val-ores erróneos.

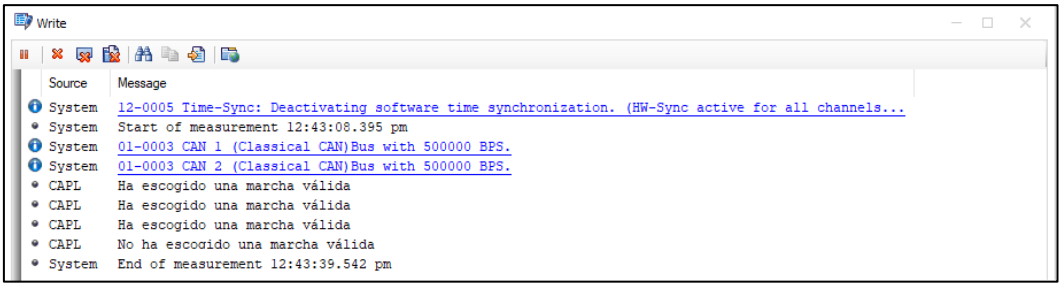


Figura 76. Mensaje de error para el calculador.

Efectivamente, la ventana write nos envía un mensaje de error, "No ha escogido una marcha valida". Además, figuran los mensajes correspondientes a las marchas que hemos engranado correctamente antes, "Ha escogido una marcha correcta". Así, podemos confirmar que **podemos pilotar el calculador con nuestra interfaz de usuario.**

Repetimos el proceso con la **bomba de aceite**. Hice una prueba introduciendo primero un régimen de giro incorrecto y luego varios regímenes coherentes: 300, 600 rpm y 900 rpm.

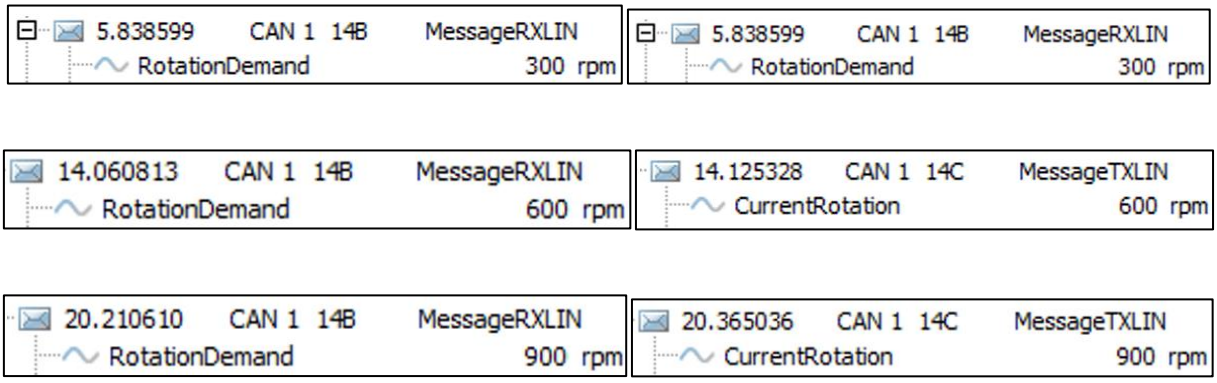


Figura 77. Verificación del pilotado de la bomba de aceite.

Podemos verificar los resultados en la ventana Trace. Podemos ver que enviamos los valores en la trama RX y el calculador nos devuelve estos valores en la trama TX.

Compruebo los resultados en la ventana "*Write*".

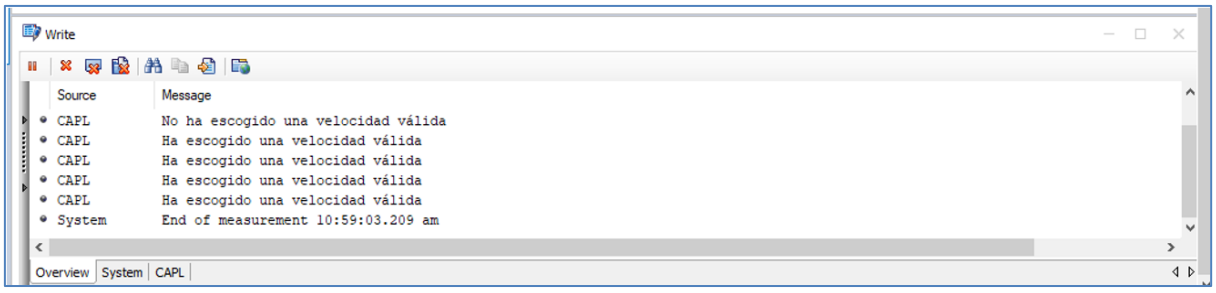


Figura 78. Mensaje de error para la bomba de aceite

De esta manera, la ventana write nos devuelve los mensajes correspondientes. Así, podemos confirmar que **podemos pilotar la bomba de aceite con nuestra interfaz de usuario.**

2 Diagnóstico de tráfico de datos

Primero de todo, realizo una **prueba en un coche prototipo**. Con tal efecto, tras haber cargado el modelo simulink correspondiente en la MUXlab, hago las conexiones necesarias entre mi ordenador portátil, la CANcase, la MUXlab y la toma OBD del coche (**véase fig. 58**).

Luego, arranco el motor y desplazo el vehículo por el aparcamiento. Al hacerlo, se intercambia información entre las ECUs del vehículo y el modelo Simulink. Entretanto, registro este intercambio de datos en un archivo .blf mediante un *Logging Block* en CANalyzer (**veáse fig. 14 y 15**).

Tras haber terminado esta prueba, **analizo detenidamente el tráfico de datos** desde mi ordenador en el laboratorio. Para esto, cargo el archivo .blf en la ventana *Measurement Setup,* ejecuto la simulación guardada espontáneamente y offline (**veáse fig. 17**) y me fijo en las muestras tomadas en la ventana *Trace*, con la opción "*Display Mode"* habilitada (**veáse fig. 19**).

Primero, analizamos el **diálogo con el calculador de la caja de cambios**. Para cada interacción, tienen que intercambiarse 3 tramas: la **consulta** del modelo Simulink al calculador (Trama DIAGNOSTIC_TOOL_to_ALL_ECUs), una **respuesta** del calculador al modelo (trama GEARBOX_to_DIAGNOSTIC_TOOL) y un **envío** del modelo al banco, que contiene la marcha engranada de la caja (trama DELIVERY_GEAR).

18.270240	CAN 1	DIAGNOSTIC_TOOL_to_ALL_ECUs	CAN Frame
18.271908	CAN 1	GEARBOX_to_DIAGNOSTIC_TOOL	CAN Frame
18.280119	CAN 2	DELIVERY_GEAR	CAN Frame
33.873051	CAN 1	DIAGNOSTIC_TOOL_to_ALL_ECUs	CAN Frame
33.874700	CAN 1	GEARBOX_to_DIAGNOSTIC_TOOL	CAN Frame
33.882924	CAN 2	DELIVERY_GEAR	CAN Frame
47.575506	CAN 1	DIAGNOSTIC_TOOL_to_ALL_ECUs	CAN Frame
47.577264	CAN 1	GEARBOX_to_DIAGNOSTIC_TOOL	CAN Frame
47.585389	CAN 2	DELIVERY_GEAR	CAN Frame
56.877180	CAN 1	DIAGNOSTIC_TOOL_to_ALL_ECUs	CAN Frame
56.879328	CAN 1	GEARBOX_to_DIAGNOSTIC_TOOL	CAN Frame
56.887061	CAN 2	DELIVERY_GEAR	CAN Frame

Figura 79. Muestras del tráfico de datos del calculador de la caja de cambios .

Podemos ver que este intercambio se ha realizado con éxito a lo largo de la prueba, de forma periódica y casi instantánea. Así, podemos confirmar que el **tráfico de datos del calculador de la caja de cambios fluye correctamente**.

A continuación, analizamos el **diálogo con la bomba de aceite**. Para cada interacción, tienen que intercambiarse 3 tramas, una **consulta** del modelo Simulink a la bomba (Trama DIAGNOSTIC_TOOL_to_ALL_ECUs), una **respuesta** de la bomba al modelo (trama PUMP_to_DIAGNOSTIC_TOOL) y un **envío** del modelo al banco, que contiene el régimen de rotación de la bomba (trama DELIVERY_ROTATION).

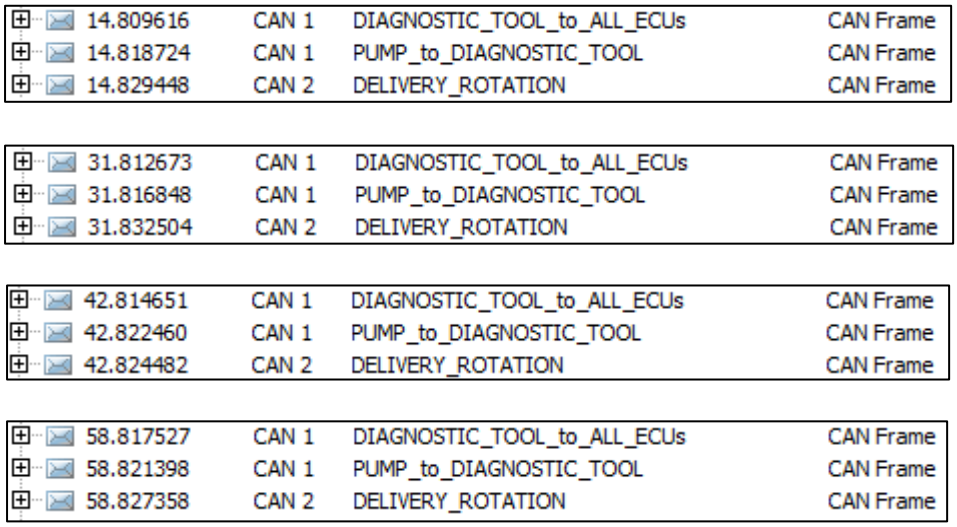


Figura 80. Muestras del tráfico de datos de la bomba de aceite.

Podemos ver que este intercambio se ha realizado con éxito a lo largo de la prueba, de forma periódica y casi instantánea. Así, podemos confirmar que el **tráfico de datos de la bomba de aceite fluye correctamente.**

3 Problemas encontrados

3.1 Orden de la secuencia

Por una parte, en el primer borrador de mi código, veía que hacía falta pulsar el botón del panel **dos veces** para hacer que la posición introducida en el panel se transfiriera a la trama enviada. Es decir, **la trama sólo enviaba la posición introducida en la pulsación anterior**, en vez de la posición de la pulsación actual.

Observe que la raíz del problema consistía en que estaba **enviando la trama antes de definirla**, de manera que realmente estaba enviando tramas definidas en la pulsación anterior. De esto, aprendemos que es importante seguir el orden correcto en el envío de tramas: **definición, envío y cancelación**. El orden erróneo que seguía antes era envío, definición y cancelación.

```
on sysvar GUI::ButtonCAN
{
  if (@GUI::ButtonCAN==1)
    {
      Preconditions();
      PressButtonCAN();
    }
}
```

Figura 81. Error, envío de trama antes de definición.

3.2 Arquitectura de señales

Por otra parte, cuando inicialicé la medición por primera vez, pude ver que, a pesar de que los scripts no daban errores de compilación y de que podía ver en la ventana Trace que se estaba enviando tramas RX al calculador, las tramas TX no experimentaban ningún cambio, es decir, **el componente del vehículo no reaccionaba**.

El problema consistía en que había omitido ciertas señales que figuraban en la documentación aportada. Di por hecho que no eran relevantes, ya que, al estudiar documentación aportada sobre interfaces de usuario similares, estas señales no se editaban ni se modificaban en los scripts aportados. Sin embargo, al incluirlas en mi código, el calculador al fin pudo operar en condiciones. Se ve que estas señales son identificativas y que el calculador sólo reacciona al recibir dichas señales.

De aquí, aprendemos que es importante **conocer la arquitectura de señales y tramas** en una base de datos y que algunas señales pueden servir una función **puramente identificativa.**

3.3 Nodo de programa

Originalmente, había creado 3 scripts y había asignado cada script a un nodo en serie. Sin embargo, he comprobado que **solamente el script asociado al último nodo activo envía tramas.**

Sería posible inhabilitar el último nodo activo, de forma que el script asociado al bloque anterior entre en vigor, pero los efectos de los otros scripts no entrarían en vigor. Es decir, solamente se puede asignar **un único nodo de programa activo**.

Hay dos formas de hacer que los efectos de todos los scripts tengan lugar simultáneamente, una es **juntando los contenidos de todos los scripts en un solo script**, y otra es haciendo que **cada script haga referencia al anterior mediante el comando** `includes`. Yo me decanté por esta última opción.

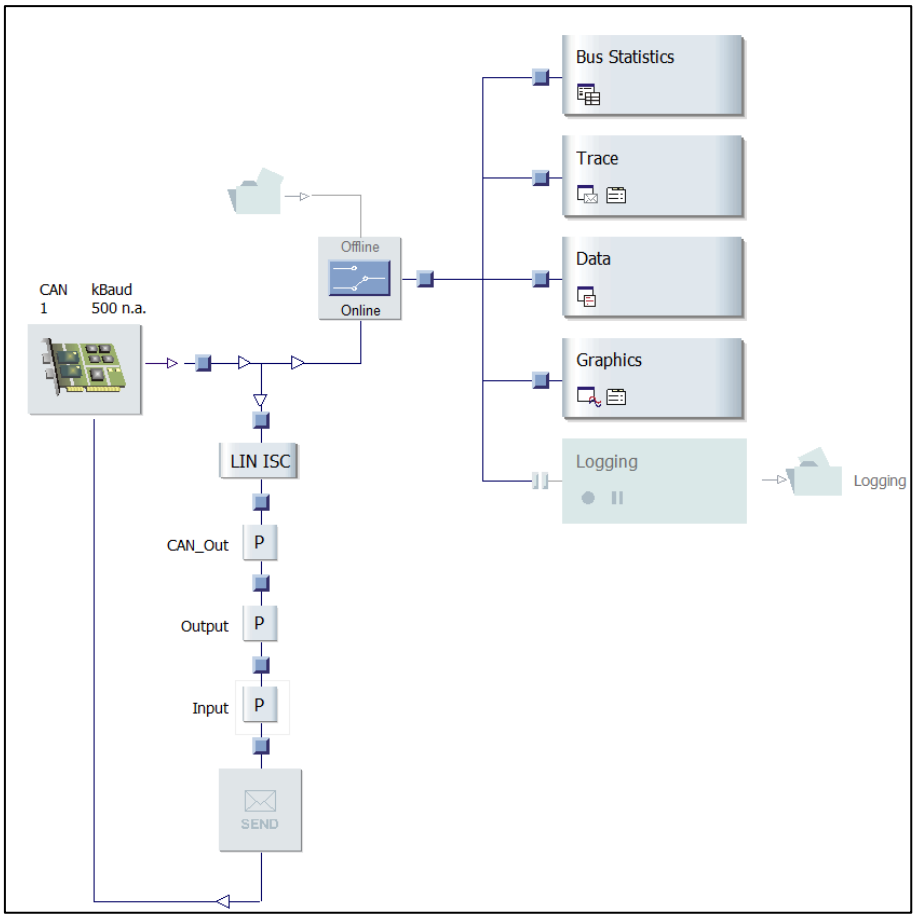


Figura 82. Error: 3 nodos de programa en serie.

4 Conclusiones

Conociendo bien las condiciones del programa CANalyzer así como los comandos y funciones del lenguaje CAPL, es posible crear interfaces que permitan al usuario pilotar calculadores del vehículo así como analizar el estado de dichos calculadores una vez implementados en el vehículo. Así, queda demostrada la eficacia de CANalyzer como software de diagnóstico de datos y de diseño de elementos gráficos.

DOCUMENTO Nº2: PLIEGO DE CONDICIONES

1 Caso de estudio: Interfaz de usuario

1.1 Software

Es necesario tener:

- La **versión 13 de CANalyzer** instalada en el ordenador.
- Las **2 bases de datos** correspondientes al bus de cada componente, **DatabaseGearbox.dbc** para el calculador y **DatabsePump.ldf** para la bomba de aceite.
- La configuración que hay que abrir desde CANalyzer, **Configuration1.cfg**, donde se puede acceder a la interfaz, así como visualizar resultados.
- Los 3 scripts de CAPL que han de ejecutarse: **CAN_out.cin, Output.can e Input.can**.
- La interfaz de usuario **Panel_TFM.xvp.**
- Fichero **variables_entorno.vsysvar**, que contiene las variables de sistema y entornos necesarios para poder ejecutar los scripts. Dichas variables ya vienen guardadas en la configuración, el fichero se adjunta como referencia por si se editan o se borran las mismas.

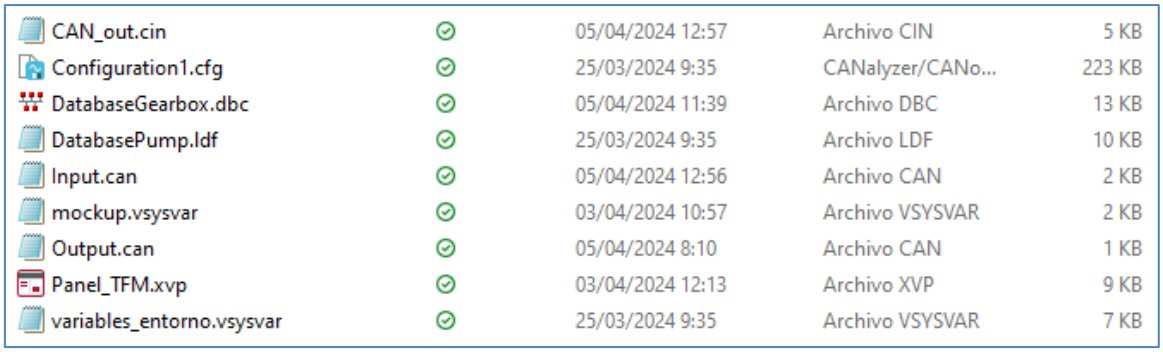


Figura 83. Documentación necesaria.

1.2 Hardware

Para empezar, necesitamos disponer tanto de

- Las 2 ECUs necesarias, **la bomba de aceite** y el **calculador de la caja de cambios**.
- Un **ordenador** donde venga instalado CANalyzer.
- Una **CANcase** y el **cableado** necesario para conectar la bomba y la ECU a nuestro ordenador.
- Una **fuente de alimentación** que proporcione tensión tanto al calculador como a la bomba de aceite para habilitar su correcto funcionamiento.

1.3 Cableado

- **Cableado de conexión CAN del calculador**

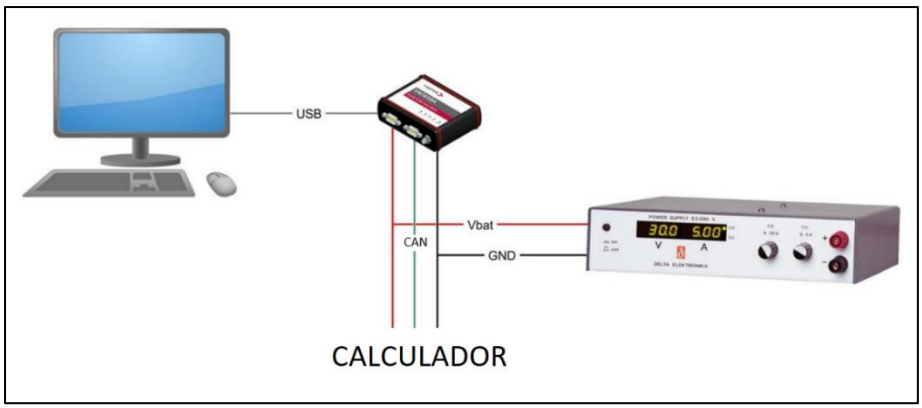


Figura 84. Cableado necesario para el calculador.

1 Conectar la CANcase al ordenador mediante un **cable USB tipo B.**

2 Conectar el calculador a una fuente de alimentación mediante **dos bananas, roja y negra**, a alimentación y a tierra respectivamente.

3 Conectar la CANcase al calculador mediante un **cable CAN**.

- **Cableado de conexión LIN de la bomba de aceite**

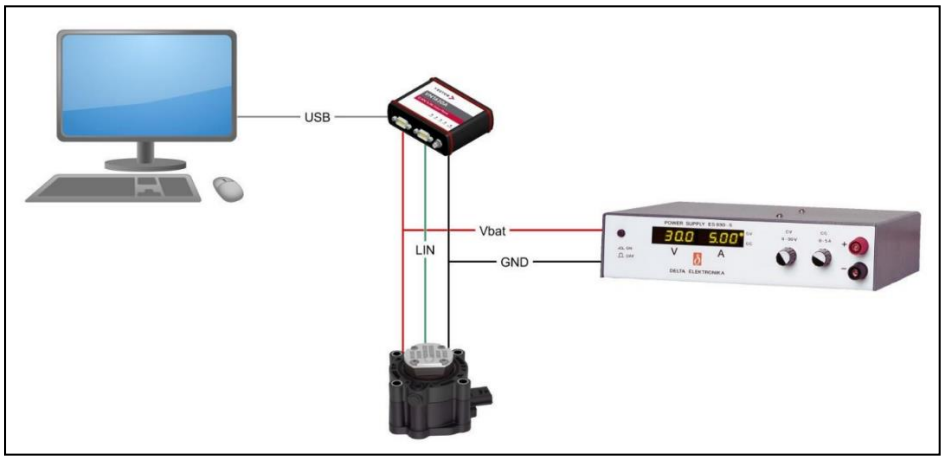


Figura 85. *Cableado necesario para la bomba de aceite.*

1. Conectar la CANcase al ordenador mediante un **cable USB tipo B.**

2. Conectar la bomba de aceite a una fuente de alimentación mediante **dos bananas, roja y negra,** a alimentación y a tierra respectivamente.

3. Conectar la CANcase a la bomba de aceite mediante un cable LIN. Es importante que el cable LIN tenga dos cables: el propio LIN y la tierra. El LIN se corresponde al pin 7 del conector DB9 y la tierra al pin 3 del conector DB9.

1.4 Licencia

Para poder usar todas las funciones de CANalyzer, es necesario disponer de la **licencia correspondiente** a la versión de CANalyzer instalada en nuestro ordenador. La licencia o bien puede estar instalada en nuestro ordenador o bien puede ir dentro de una CANcase conectada al ordenador.

Sin una licencia de CANalyzer, no es posible inicializar mediciones, importar o convertir archivos .lbf ni guardar configuraciones. Por lo tanto, no conviene modificar exhaustivamente las configuraciones sin tener licencia, ya que el progreso realizado no se guardaría.

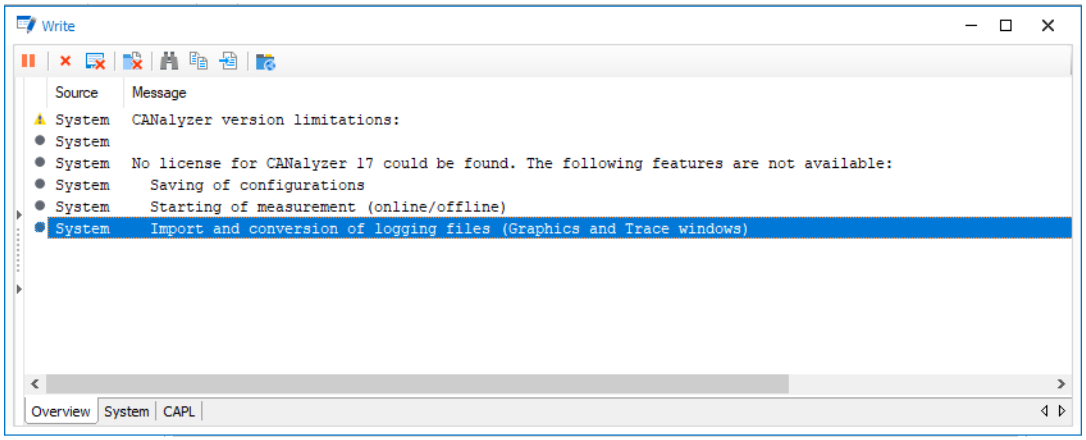


Figura 86. Versión limitada de CANalyzer.

2 Caso de estudio: Diagnóstico de Tráfico de datos

2.1 Software

Es necesario tener:

- La **versión 13 de CANalyzer** instalada en el ordenador.
- Las **2 bases de datos** correspondientes al bus de cada componente, para la toma OBD del coche **DEBUG_TOOL_to_OBD.dbc** y para la MUXlab **DEBUG_TOOL _to_BENCH.dbc**
- La configuración que hay que abrir desde CANalyzer, **Configuration2.cfg**, donde se puede acceder a la interfaz, así como visualizar resultados.

Figura 87. *Cableado necesario para la toma OBD del coche.*

2.2 Hardware

Necesitamos disponer de:

- Una **MUXlab** donde esté cargada el modelo simulink necesario.
- Un **ordenador** donde venga instalado CANalyzer.
- Un **vehículo** para realizar la prueba.
- Una **CANcase** y el **cableado** necesario para conectar la toma OBD del vehículo y la MUXlab a nuestro ordenador y entre sí.

2.3 Cableado

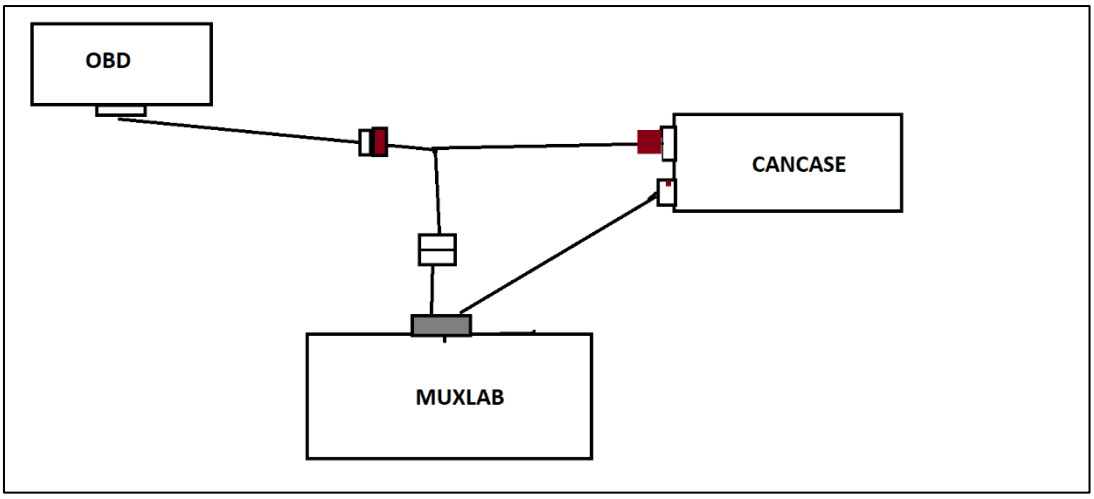


Figura 88. *Cableado necesario para la toma OBD del coche.*

1. Conectar la CANcase al ordenador mediante un **cable USB tipo B**.

2. Mediante **una "Y", confeccionada en taller a partir de dos cables CAN, y dos cables CAN adicionales**, conectar la toma OBD del coche a la CANcase y a la MUXlab simultáneamente.

3. Conectar la MUXlab a la CANcase mediante un **cable CAN**.

4. Conectar la MUXlab al mechero del coche, para alimentarla, mediante dos bananas, **roja y negra**.

DOCUMENTO Nº3: PRESUPUESTO

1 Pilotado de ECUs mediante interfaz de usuario

Llevamos la cuenta del precio del software y hardware necesarios para llevar a cabo nuestro proyecto. Para poder usar los **productos de la compañía Vector Informatik GmbH**, tendremos que solicitar un presupuesto desde la página web de la empresa.

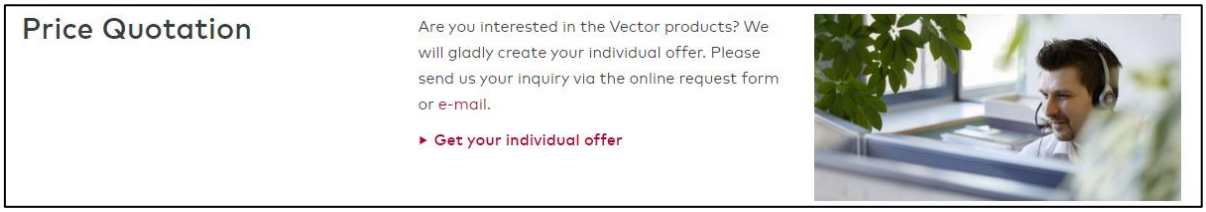


Figura 89. Presupuesto de productos de Vector [5]

Los precios negociados serían los siguientes:

- CANcase VN1630A: 2.000€.
- Licencia CANalyzer 13: 4.500€.

Por otra parte, tenemos los **calculadores** solicitados por el cliente:

- Bomba de aceite: 50€.
- Calculador de la caja de cambios: 100€.

Finalmente, tendremos **componentes de taller**, los cuales la empresa tendría que solicitar desde el portal RS Componentes [6]:

- Fuente de alimentación: 230€.
- Cableado (1 cable USB tipo B, 1 cable CAN, 1 cable LIN, dos bananas rojas y dos bananas negras): 3€ + 20€ + 10€ + 40€ = 73€

Así, el **presupuesto de este caso de estudio** asciende a 6.953€

2 Diagnóstico de tráfico de datos

Para este caso de estudio, ya dispondremos de la licencia de CANalyzer 13, la CANcase, un cable USB tipo B y un cable CAN del caso anterior.

Por otra parte, requeriremos, a nivel de software, una **licencia de Matlab**, para poder verificar que el modelo Simulink funciona correctamente y luego compilarlo. A nivel hardware, necesitamos una **MUXlab y el cableado necesario**.

- Licencia de MATLAB para un usuario: 2.000€
- MUXlab6: 1.300€
- Cableado (una "Y", dos cables CAN, bananas roja y negra): 40€+ 40€ + 20€= 100€

Después, realizaremos la prueba en un prototipo de coche prestado por la empresa.

Por lo tanto, el **presupuesto para este caso de estudio** se ajusta a 3.400€.

En conclusión, **el presupuesto total de ejecución material para nuestro proyecto** engloba 10.353€.

Resumen	**Importe en Euros**
CANcase VN1630A	2.000,00
Licencia CANalyzer 13	4.500,00
Bomba de aceite	50,00
Calculador de la caja de cambios	100,00
Fuente de alimentación	230,00
Cableado (1 cable USB tipo B, 1 cable CAN, 1 cable LIN, dos bananas rojas y dos bananas negras)	73,00
Licencia de MATLAB para un usuario	2.000,00
MUXlab6	1.300,00
Cableado (una "Y", dos cables CAN, bananas roja y negra)	100,00
Presupuesto de Ejecución Material	**10.353,00**
Gastos generales (10%)	1.035,30
Beneficio industrial (20%)	2.070,60
Total	**13.458,90**
I.V.A (21%)	2.826,37
Presupuesto Total	**16.285,27**

Aplicándose al anterior presupuesto de ejecución material el **10% de gastos generales** en concepto de diseño y horas de trabajo y un **20% de beneficio industrial**, se obtiene un presupuesto sin I.V.A de 13.458,90€. Aplicando al anterior presupuesto el I.V.A del 21% se obtiene un presupuesto total de 16.285,27 €.

DOCUMENTO Nº4: ANEXOS

Este apartado engloba solamente el código fuente empleado en el primer caso de estudio, **el pilotado de ECUs mediante interfaz de usuario**, puesto que no he tenido que desarrollar scripts nuevos en el segundo caso de estudio, el diagnóstico de tráfico de datos.

1 Script CAN_OUT.cin

```
Variables
{
  //Definición de tramas RX
  message CAN1.MessageRXCAN msgMessageRXCAN;
  linframe MessageRXLIN msgMessageRXLIN;

  //Definición de temporizadores
  msTimer tmrMessageRXCAN;
  msTimer tmrMessageRXLIN;
}

//Transferencia de valores de las variables de sistema del entorno PC a las señales en las tramas RX
export void WriteMessageRXCAN()
{
  msgMessageRXCAN.ModeDemand.phys = sysGetVariableInt(sysvar::PC::ModeDemand);
  msgMessageRXCAN.GearDemand.phys = sysGetVariableInt(sysvar::PC::GearDemand);
}

export void WriteMessageRXLIN()
{
msgMessageRXLIN.RotationDemand.phys = sysGetVariableInt(sysvar::PC::RotationDemand);
}

//Envío cíclico de tramas mediante temporizadores
on timer tmrMessageRXCAN
{
   output(msgMessageRXCAN);
   setTimer(tmrMessageRXCAN, 10 + random(2));
}
```

```
on timer tmrMessageRXLIN
{
  output(msgMessageRXLIN);
  setTimer(tmrMessageRXLIN, 10 + random(2));
}
```

2 Script Output.can

```
//Referencia al script CAN_out.cin
includes
{
  #include “CAN_out.cin”
}

variables
{
}

//Función para la inicialización de temporizadores
export void Preconditions()
{
  setTimer(tmrMessageRXCAN, 10); //10 ms
  setTimer(tmrMessageRXLIN, 5); //5 ms
}

//Función para la cancelación de temporizadores
export void Postconditions()
{
  cancelTimer(tmrMessageRXCAN);
  cancelTimer(tmrMessageRXLIN);
}
```

3 Script Input.can

```
//Referencia al script output.can
includes
{
  #include “output.can”
}

variables
{
  msTimer tmrButton_Aux;
}

//Evento de pulsación de botones
on sysvar GUI::ButtonCAN
{
  if (@GUI::ButtonCAN==1)
    {
       PressButtonCAN();
    }
}

on sysvar GUI::ButtonLIN
{
  if (@GUI::ButtonLIN==1)
    {
       PressButtonLIN();
    }
}

//Definición de temporizador auxiliar para detener temporizadores
on timer tmrButton_Aux
{
  PostConditions();
}
```

```
// Transferencia de valores de las variables de sistema del entorno GUI a variables
de sistema del entorno PC
Void PressButtonCAN()
{
  If(@GUI::InputCAN<0 || @GUI::InputCAN>6
    {
      Write("No ha introducido una marcha correcta")
    }
  Else
    {
      sysSetVariableInt(sysvar::PC::ModeDemand, 1)
      sysSetVariableInt(sysvar::PC::GearDemand, @GUI::InputCAN)

      writeMessageRXCAN();
      Preconditions();
      setTimer(tmrButton_Aux, 300);
      write("Ha introducido una marcha correcta")
    }
}

Void PressButtonLIN()
{
  If(@GUI::InputLIN<0 || @GUI::InputLIN>1200
    {
      Write("No ha introducido un valor correcto")
    }
  Else
    {
      sysSetVariableInt(sysvar::PC::RotationDemand, @GUI::InputLIN)

      writeMessageRXLIN();
      Preconditions();
      setTimer(tmrButton_Aux, 300);
      write("Ha introducido un valor correcto")
    }
}
```