Curso Académico:

ÍNDICE DE DOCUMENTOS

- *Memoria*
- *Presupuesto*
- *Anexo 1*
- *Anexo 2*

Resumen

El conocimiento de las distintas ingenierías como materias aisladas lleva a una conclusión sesgada y a la obtención de unos resultados limitados. Es precisamente la interacción entre las mismas la que aporta un potencial inmenso. En esta tesis queda de manifiesto dicha interacción entre tres ramas de la ingeniería en distinto estado de maduración, la automática, la informática y la eléctrica.

En este proyecto, se ha puesto en marcha un banco de pruebas de servoaccionamientos síncronos unidos entre ellos por su eje, actuando uno como motor y otro como carga. La operación de estos se realiza de forma remota a través de una aplicación móvil que permite el control del sistema de servoaccionamientos y la monitorización de la respuesta del sistema frente a diferentes entradas.

Aun estando las disciplinas citadas en distinto estado de evolución, coordinándolas adecuadamente, se pretende conseguir un control efectivo a través de una interfaz de usuario sencilla e intuitiva.

Para ello, el proyecto se basará en tres pilares fundamentales. El circuito de potencia, la aplicación de control, a través del software de automatización de B&R, y la aplicación para dispositivo móvil, para la que se hará uso del software de Android Studio.

Abstract

Understanding the different engineering fields as specific subjects leads to biased conclusions and limited results. Thus, is the interaction between them what provides an immense potential. In this thesis, this interaction will be seen between three fields of engineering, automatics, computer, and electrical engineering.

The objective is the development and setup of a test bench of synchronous servo drives, connected one to the other by its, one acting as a motor and the other as a load. The operation of these is carried out remotely through a mobile application that allows the control of the servo drive system and the reading of the response of the system against different inputs.

Even though the disciplines mentioned before are in a different state of their evolution, properly coordinated, they are intended to achieve effective control through a simple and intuitive user interface.

For this, the project will be based on three fundamental pillars. The power circuit, the control application, through the B&R automation software, and the application for mobile devices, for which the Android Studio software will be used.

TRABAJO FINAL DE GRADO

Valencia. 2019/2020 Iván Arenas Kelbelova

MEMORIA

DISEÑO Y DESARROLLO DE UN BANCO DE PRUEBAS DE SERVOACCIONAMIENTOS DE 4,5kW CON OPERACIÓN REMOTA

Índice de memoria

1. OBJETIVOS DEL TRABAJO 1

2. INTRODUCCIÓN AL PROBLEMA: ANTECEDENTES, MOTIVACIÓN Y JUSTIFICACIÓN 2

2.1. Antecedentes 2

2.2. Motivación 2

2.3. Estructura de la memoria 3

3. NORMATIVA 4

4. MONTAJE FÍSICO 5

4.1. Introducción 5

5. Programa AUTOMATION STUDIO. 9

5.1. Introducción 9

5.2. Estructura del proyecto de Automation 9

5.2.1. Orden App-Motor 10

5.2.2. Datos Motor-App 11

5.3. Automation Studio – Preparación del proyecto 11

5.3.1. Introducción al software 11

5.3.2. Preparación de la Physical View 13

5.3.3. Dynamic Node Allocation 15

5.3.4. Configuración IP 16

5.3.5. Librerías 18

5.3.6. Introducción de programas y velocidad de ciclo 18

5.3.6. Ajuste del controlador, primer arranque y errores en primera puesta en marcha 19

5.4. TCP server 20

5.4.1. Inicialización y etapas 21

5.4.2. Recepción de datos. 22

5.4.3. Envío de datos. 23

5.5. Motion modules 24

5.5.1. Motion General 24

5.5.2. Motion 1 25

5.5.2.1. Caso común 25

5.5.2.2. Caso Power 26

5.5.2.3. Cambio de Velocidad 28

5.5.2.4. Velocidad de eje 29

5.5.3. Motion 2 29

5.5.3.1. Aplicación de par 29

5.5.3.2. Lectura de par 31
5.6. Main Program 31
5.6.1. Señales de habilitación 32
5.6.1.1. Enable, Disable y ActionDone. 33
5.6.2. Adaptación de par y velocidad de entrada 34
5.6.2.1. LReceivedData 35
5.6.3. Adaptación de datos de salida 36
6. APLICACIÓN ANDROID 37
6.1. Introducción a Android 37
6.1.1. Android 37
6.1.2. Android Studio 37
6.2. Introducción a las partes del proyecto 38
6.3. Preparación del entorno 40
6.3.1. Gradle 40
6.3.2. Values 40
6.3.3. Drawable 41
6.3.4. Manifest 41
6.4. Interfaz de usuario 41
6.5. Gestor de fragmentos 44
6.5.1. Definiciones 44
6.5.3. Gestor de fragmentos 46
6.6. Fragmento 1 – InitFragment(…) 47
6.6.1. onCreateView(…) 47
6.6.2. seekBar.setOnSeekBarChangeListener 47
6.6.3. OnClick(…) 48
6.6.4. stateConnection 49
6.7. Fragmento 2 – VisualDataFragment() 50
6.7.1. OnCreateView(…) 50
6.7.2. PlotData 50
6.7.3. OnClick(…) 51
6.8. TCPProcess - StartClient 52
6.8.1. Conexión y recepción de datos 52
6.8.2. Envío de datos 53
6.9. SharedViewModel 53
7. CODIGOS DE COMUNICACIÓN 56
7.1. Introducción 56

7.2. Códigos de orden 56
7.3. Códigos de datos 58
8. MANUAL DE ARRANQUE 59
8.1. Introducción 59
8.2. Interpretación de señales 59
8.3. Conexión App – Banco de ensayos 60
8.4. Puesta en marcha – Apagado 60
8.4.1. Puesta en marcha 61
8.4.2. Apagado 62
8.4.3. Inconvenientes y recomendaciones 62
8.5. Graficado 62
9. CONCLUSIÓN Y PROPUESTAS DE MEJORA 64
10. BIBLIOGRAFÍA 65

Índice de ilustraciones

Ilustración 1. Representación esquemática del montaje físico. Fuente propia. En la imagen se muestran los distintos elementos del montaje físico, con su respectiva numeración y conexiones con el marcaje del número de líneas. 5
Ilustración 2. Comunicación entre los módulos del proyecto desarrollado en Automation Studio. Fuente propia. La imagen muestra esquemáticamente la relación entre los distintos módulos del proyecto y la función básica que ejecutan.......... 10
Ilustración 3. Captura general del programa de Automation Studio. Fuente propia. Se detallan las distintas partes en las que se divide el entorno de trabajo de este software y se describen a continuación.......... 12
Ilustración 4. Capturas del Harware catalag y sección de selección de motores. Fuente propia.... 13
Ilustración 5.Vista del System Designer de Automation Studio. Fuente propia. En esta se visualizan el PLC, los dos servoaccionamientos y los motores conectados entre sí. 14
Ilustración 6. Captura de la sección de configuración del DNA. Fuente propia. En la imagen se muestra el apartado de configuración DNA con las características del primer motor de línea. 15
Ilustración 7. Captura de la sección Online settings de Automation Studio. Fuente propia. Se muestra la IP configurada junto el uso en configuración activa para forzar el uso de dicha IP. 16
Ilustración 8. Captura del apartado de configuración del PLC. Fuente propia. 17
Ilustración 9. Captura del fichero Cpu.sw y carpeta de librerías. Fuente propia.......... 18
Ilustración 10. Captura del fichero Cpu.sw sección de ciclos de funcionamiento. Fuente propia. En esta se visualizan los distintos módulos funcionando en sus ciclos de funcionamiento marcados. 19
Ilustración 11. Esquema de funcionamiento de la estructura CASE del módulo TCP en Automation Studio. Fuente propia. En esta se muestran las etapas por las que pasa el módulo TCP para gestionar de forma correcta la conexión de dispositivos y el envío y recepción de datos.......... 21
Ilustración 12. Esquema de funcionamiento de la estrucutra CASE en los módulos de Motion. Fuente propia. Ilustración similar a la ilustración 13 pero para la plantilla del módulo motion..... 24
Ilustración 13. Fichero de declaración de nuevos tipos. Fuente propia. En la imagen se muestra el fichero Motion1.typ que permite la declaración de nuevos tipos de variables. 27
Ilustración 14.Captura de la sección I/O mapping. Fuente propia. Permite seleccionar variables a salidas del PLC de forma que reflejen su estado. 32
Ilustración 15. Captura del código de gestión de acciones del Motion 1. Fuente propia. Marca el intervalo donde en caso de producirse un cambio en la variable ActionDone se podría producir un error. 33
Ilustración 16. Captura del código de gestión de datos de velocidad y par del MainProgram en Automation Studio. Fuente propia. 35
Ilustración 17. Imagen de la etapa de diseño de la aplicación. Fuente propia. En esta imagen se muestra una vista esquemática de ambos fragmentos, mostrando las funciones que se pretende implementar en cada área. 39
Ilustración 18. Vista de ambos fragmentos de la aplicación Android. Fuente propia. En esta se visualizan los dos fragmentos de la aplicación. En la vista izquierda se muestra el fragmento de acciones, mientras que a la derecha se muestra un ejemplo de representación gráfica del fragmento de visualización de datos. 42
Ilustración 19. Relación de actividad principal y fragmentos de la aplicación Android. Fuente propia. En esta se muestran las relaciones entre los distintos módulos. 45

Ilustración 20. Muestra del flujo de datos de estado del motor entre fragmentos. Fuente propia. El flujo entra por el fragmento 1 y para pasar al fragmento 2 es necesario el uso de una interfaz ViewModel que permita el intercambio de datos entre ambas. 54
Ilustración 21. Uso de los bits de comunicación. Fuente propia. Se detallan los distintos bits de la palabra pasada y su uso. 57
Ilustración 22. Displays de estado del servoaccionamiento. Imagen de la guía de ayuda sección Hardware/ Motion Control/ ACOPOS P3/ 8EI servodrives / Status indicators. 59
Ilustración 23. Cambio de estados y señalización en el servoaccionamiento. Muestra el flujo entre estados de un servoaccionamiento durante su encendido. 61

1. OBJETIVOS DEL TRABAJO

El objetivo principal de este proyecto es el diseño, desarrollo y puesta a punto de un banco de pruebas de servomotores síncronos con control remoto. El sistema constará de dos servomotores unidos por su eje, ejerciendo uno un par motor y el otro un par resistivo, con esta disposición se pretende simular las distintas situaciones a las que se puede someter a un motor en carga y la respuesta de éste. Al tratarse de un montaje que se ajusta a una situación real común es idóneo para la realización de distintas pruebas en los ámbitos académico y de investigación. El desarrollo de la interfaz de control remoto se hará a través del uso del software de Android Studio para desarrollar una aplicación móvil que permita tanto el manejo de los motores como la recepción de información procedente de los mismos. Además, para la unión entre ambos sistemas programaremos un PLC con el uso del software de Automation Studio. Dicho PLC hará la función de procesar y transmitir las instrucciones dadas desde la aplicación móvil y retornar valores de par resistente aplicado y velocidad de los motores

Con el fin de lograr el objetivo principal previamente mencionado se plantean los siguientes objetivos parciales:

- Montaje del sistema eléctrico de servomotores con los complementos adecuados y conexión a red eléctrica.
- Programación del entorno de control de dichos servomotores con el software de Automation Studio que permita la habilitación y el manejo de estos, además de la obtención de datos.
- Programación de la interfaz de usuario mediante una app sencilla e intuitiva con el uso de Android Studio.
- Montaje y configuración de la vía de comunicación TCP y desarrollo del protocolo de comunicación que permita la comunicación de los dispositivos móviles con el PLC.

Además, con objeto de facilitar el uso formativo de dicho sistema, se realizará un apartado que incluye un manual básico de arranque, con ciertas indicaciones que se deben tener en cuenta para conocer el estado del motor. Y los pasos básicos a seguir para el encendido y apagado de los motores de forma idónea.

2. INTRODUCCIÓN AL PROBLEMA: ANTECEDENTES, MOTIVACIÓN Y JUSTIFICACIÓN

2.1. Antecedentes

Un motor no es sino una máquina eléctrica rotativa que actúa como un convertidor bidireccional, energía mecánica – energía eléctrica. La diferencia principal entre un motor y un servomotor, es que el segundo tiene un sistema de control, un sistema de regulación específicamente diseñado para tareas de posicionamiento; esto permite un control del movimiento, velocidad y par aplicados y un retorno de datos como la posición o la velocidad. Por tanto, para centrar el problema, se hace referencia desde la estructura más simple entendida como máquina eléctrica rotativa hasta llegar a la definición de servomotor síncrono 'brushless'.

Los motores Brushless DC tienen una gran penetración en el mercado dada su idoneidad para aplicaciones de movilidad personal, además del interés que desde hace décadas han tenido en el área de automatización industrial por su sencillo control por comparación con otras máquinas rotativas.

Una tendencia actual en los fabricantes de accionamientos es trasladar las funciones más elevadas del control hacia elementos separados del habitual convertidor de frecuencia, a menudo de tipo PLC, lo que permita una fácil integración del accionamiento y de su control en niveles jerarquizados de control de la planta.

Por último, todo el mercado se mueve en la dirección marcada por la industria 4.0, con soluciones integradoras en las que la disponibilidad de datos y de sistemas de operación separados de los niveles más bajos de control, integrados con comunicaciones industriales y enlazados con dispositivos móviles para ser utilizados como interfaz de usuario de gran versatilidad.

2.2. Motivación

Analizadas las tendencias de la innovación tecnológica y el interés de la industria por las aplicaciones 4.0, se ha considerado oportuno poder contar en los laboratorios de Ingeniería Eléctrica con un banco de ensayos que recogiera los elementos más significativos de esas tendencias.

El laboratorio de la unidad docente de máquinas eléctricas del DIE cuenta con varios bancos de ensayos de diferentes tipologías de máquinas, pero, la irrupción de los servomotores 'brushless' DC hacen muy interesante contar con un banco específico para este tipo de máquinas, con tecnología de última generación.

Alrededor del interés que generan los servomotores 'brushless' como tecnología emergente, surge el interés de aplicar otras tendencias también mencionadas en el apartado anterior. Por ello, se

busca la separación del control de los servoaccionamientos, que utilice un único elemento de control, centralizando el manejo de los distintos servoaccionamientos.

Además, se ha querido seguir la tendencia mencionada sobre la separación de los niveles más bajos de control de una interfaz de usuario. Dicha motivación implica la necesidad de un establecimiento de comunicación entre el banco físico y el dispositivo móvil que pretende tener el manejo.

2.3. Estructura de la memoria

La memoria se encuentra estructurada en distintos módulos que tienen una función precisa dentro del sistema. Durante el desarrollo de dichos módulos se comenta cómo se ha llevado a cabo el desarrollo de cada módulo, la función de las partes fundamentales y la relación con el resto de módulos. A continuación, se da una descripción de los apartados fundamentales.

- Montaje físico: En este módulo se desarrolla el montaje físico realizado. Se detallan los distintos elementos que conforman el banco de motores y las conexiones que se realizan entre ellos.

- Programa Automation Studio. Programa cargado en el PLC de la parte física, está encargado del control de los motores y de la comunicación con la interfaz de usuario. Se dividirá en cuatro módulos que se encargan de funciones específicas dentro del sistema.

- Aplicación Android Studio. Conforma la interfaz de usuario, esta permite la comunicación con el PLC, no solo para la ejecución de órdenes, sino, también, para la recepción y muestra de datos.

Los apartados mencionados anteriormente son los pilares fundamentales del proyecto. A continuación, se mencionan otros apartados que, aunque no conforman la base del proyecto, son relevantes para su funcionamiento como conjunto y uso.

- Códigos de comunicación: En dicho apartado se detalla el formato en el que se envían y reciben las órdenes para que la comunicación entre el PLC y la aplicación Android sea satisfactoria.

- Normativa: Dicho apartado no tiene mucho peso en el proyecto ya que los manuales de cada elemento especifican las necesidades de conexión y uso. Aun así, en dicho apartado se han añadido las normas que se deben seguir en estos tipos de instalaciones.

- Manual de arranque: En dicho apartado se detalla la secuencia correcta de cómo se debe realizar el arranque y paro del banco de ensayos. Además, se explican ciertos indicadores que permiten al usuario conocer el estado en el que se encuentra cada dispositivo.

3. NORMATIVA

Aunque no aparece con un gran peso dentro de este proyecto, en todos los ámbitos donde se haga uso de máquinas eléctricas es necesario el uso de ciertos reglamentos que reducen al mínimo los posibles errores y accidentes y ayudan a proteger los dispositivos involucrados.

Las normativas con las que interactúa este proyecto incluyen:

- Reglamento electrotécnico para baja tensión. Este regula las condiciones técnicas y garantías que deben cumplir las instalaciones eléctricas que se encuentran conectadas a la red de baja tensión. De este obtendremos las mínimas normas necesarias en las que debe encontrarse nuestra instalación
- Instalación de receptores. Motores. ITC-BT-47. En particular también se encuentra dentro la instrucción técnica 47 que hace referencia a los requisitos de instalación de los motores.

Además de estas normativas de ámbito nacional, se seguirán las recomendaciones marcadas en los manuales de los elementos utilizados, que se refieren a dirección de montaje, condiciones de aireado, dimensión de los conductores, y otros factores que se consideran relevantes para garantizar la protección tanto de la instalación como de las personas.

En el ámbito de la programación en automatización industrial también es relevante mencionar que el proyecto se basará en la norma de la industria IEC 61131-3. Este es un apartado de la norma IEC 61131 que tiene como finalidad estandarizar los autómatas programables. El documento 3 de esta norma define los estándares respecto a semántica y estructura de los lenguajes a utilizar. Definiendo dos lenguajes gráficos (LD y FBD) y dos de texto (ST y IL).

4. MONTAJE FÍSICO

4.1. Introducción

El montaje físico se refiere al montaje del banco de ensayos, y de la unión de los distintos elementos que lo conforman. A continuación, se observa una imagen que esquematiza el montaje realizado. Es importante mencionar que con esta imagen se pretende dar una noción de los elementos principales del banco de pruebas, no muestra todos los puertos de los elementos, solo muestra los que intervienen en las conexiones.

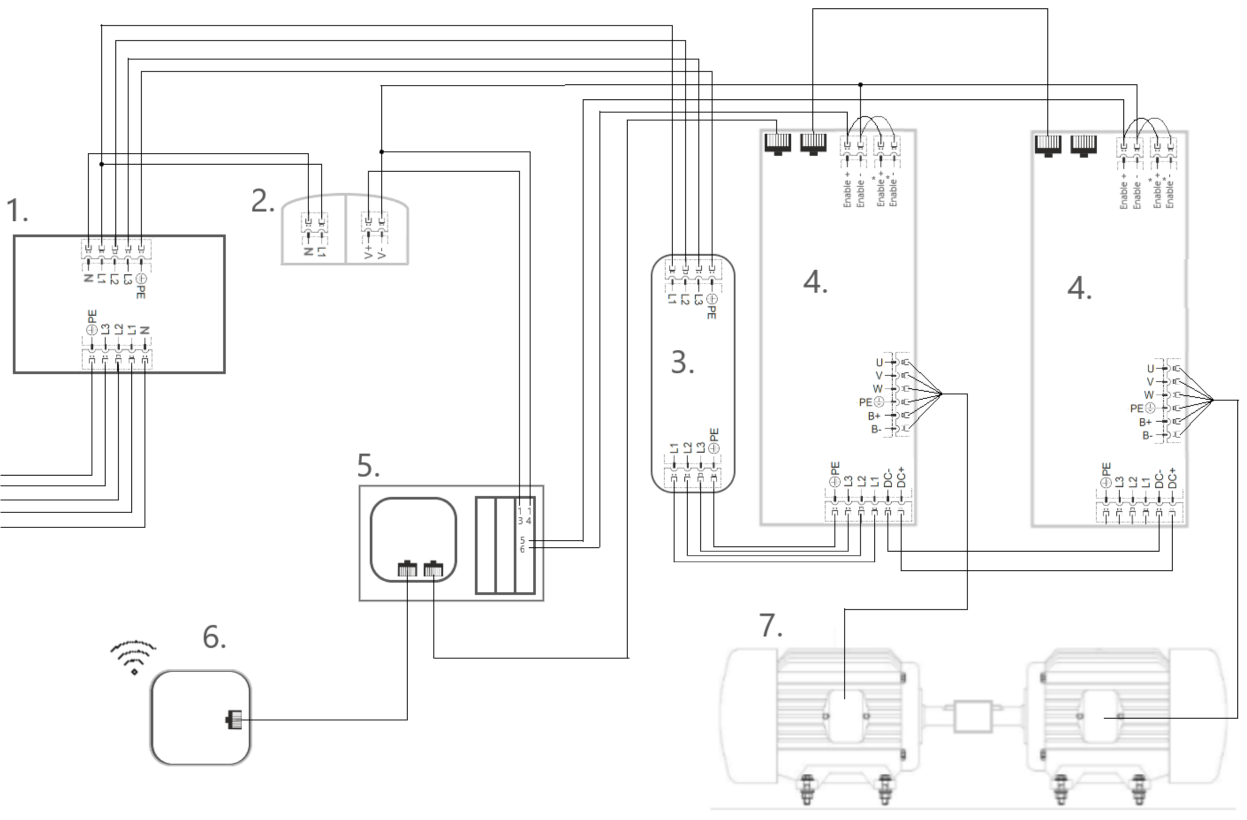


Ilustración 1. Representación esquemática del montaje físico. Fuente propia. En la imagen se muestran los distintos elementos del montaje físico, con su respectiva numeración y conexiones con el marcaje del número de líneas.

A continuación, se aclara qué es cada elemento, características, función dentro del banco de ensayos y relación en el resto de elementos del sistema:

1. Protección (Interruptor automático).

Cumple la función de protección del sistema y de las personas ante errores en las conexiones que puedan dar lugar a peligrosas tensiones y corrientes, con el fin de evitar posibles accidentes o daños en el equipo. En caso superar los rangos de funcionamiento

normal, este corta las 3 líneas de alterna más el neutro de forma segura y permitiendo reconectar a red una vez solucionada la irregularidad. Además, este permite la conexión y desconexión a red manual en caso de ser necesaria.

Consta de una entrada y una salida de tres líneas y un neutro, además salen una línea y un neutro que alimentan la fuente de 24 voltios.

2. Fuente de 24 V

Se trata de una fuente de 24 voltios DC, en la cual entra una línea y el neutro al salir del interruptor diferencial y se convierte a una salida de continua de 24 V y un neutro. La función de esta fuente de alimentación es alimentar el PLC y a los servoaccionemientos por sus entradas habilitadas para ello.

3. Filtro pasivo de línea. 16 A, 3x 480 VAC, 50/60 Hz, IP20.

Aunque no siempre es necesario el uso de un filtro de armónicos en las instalaciones de motores eléctricos, en lo que se refiere a los productos de B&R está bastante extendido su uso. El concepto de funcionamiento de un filtro pasivo es bastante simple, se basa en el uso de una derivación con una carga LC. Esta derivación absorbe armónicos de la frecuencia deseadas, siendo posible el filtrado de más de una con la colocación de distintas derivaciones en paralelo. Con este filtro pasivo se pretende eliminar los posibles armónicos que tienen su origen en la red y se evita introducir nuevos a la misma.

Por la entrada conecta con el interruptor automático y constituye la entrada del suministro de potencia al accionamiento trifásico (ACOPOS P3), que permite el manejo adecuado de los motores. Además da voltaje 0 a una de las entradas del Enable de los servoaccionamientos.

4. Accionamientos trifásicos. Servoaccionamiento ACOPOS P3. Modelo 8EI013HWS10.0600-1.

Un servoaccionamiento actúa como un elemento de control, que permite controlar la velocidad, el par y la posición; además de la lectura de esta última con alta precisión. Estas características son idóneas para poder tener un control completo sobre las operaciones de un servomotor.

Para llevar a cabo estas operaciones los servoaccionamientos constan de una unidad de control y una de potencia. La unidad de potencia se encarga de amplificar la señal dada por la unidad de control, con esta variación en la amplitud del voltaje se lleva a cabo un control de los distintos parámetros de los servomotores. Por otro lado, la unidad de control también está encargada de leer los datos proporcionados por el encoder que lleva incluido el motor, con el fin de controlar la posición en todo momento. En este caso como se ha mencionado arriba se trata de modelo ACOPOS P3 de la marca B&R. Está habilitado para un único eje por lo que usaremos uno por servomotor.

Dentro del sistema del banco de ensayos tienen un sistema de conexiones distintos para cada servomotor. Para el resto del documento, si se menciona servoaccionamiento 1, se

está haciendo referencia al que maneja al motor que hace esfuerzo mecánico. Sin embargo, si se menciona el servoaccionamiento 2, se estará haciendo referencia al que gestiona el motor que ejerce un par resistente.

El servoaccionamiento 1 recibe la entrada de potencia proveniente del filtro a 330V en alterna, además, recibe otro suministro de potencia de corriente continua proveniente del servomotor 2 (cuya proveniencia se explica más adelante). Por otro lado, la potencia suministrada es manipulada y cedida a su servomotor correspondiente. Además de las entradas de potencia, el servoaccionamiento 1 tiene una entrada en formato PowerLink PLK, por el que intercambia señales y datos con el PLC, y una salida del mismo formato que hace de puente entre el PLC y el servoaccionamiento 2. Por último, el servoaccionamiento tiene dos conexiones digitales que corresponden a una señal digital del PLC (una de la señal de 5V) y una señal de referencia a tierra proveniente de la fuente, por las que se maneja la señal habilitación 'enable' que habilita el funcionamiento de dichos servomotores, y que en caso de ser desconectada corta el suministro de potencia a los mismos.

El servoaccionamiento 2 tiene una entrada de potencia alterna a través de la conexión con su servomotor que está actuando en modo generador, y una salida de potencia continua que pasa la energía sobrante del generador al servoaccionamiento 1. Además, como se ha mencionado anteriormente tiene una conexión PLK proveniente del servoaccionamiento 1 (que actúa de puente con el PLC) con el que se intercambian datos para el control. Por otro lado, incluye la entrada analógica de otra señal 'enable' proveniente del PLC que habilita y deshabilita su funcionamiento.

Cabe mencionar la existencia de que existen dos enables por servoaccionamiento, esto se usa en caso de que exista una segunda dependencia de funcionamiento, en este caso se encuentran puenteados.

5. PLC. X20CP1382.

Se conoce como PLC o controlador lógico programable, a un dispositivo electrónico capaz de cargar y ejecutar un programa que habitualmente dispone de numerosos puertos de entrada y salida de datos que le permiten recibir señales y emitir señales de control en respuesta a las anteriores. Son elementos de uso muy extendido en la industria y son los encargados del control de distintos tipos de procesos.

En este caso disponemos de un PLC de la clase X20CP, se trata de un modelo con 30 entradas/salidas digitales y 2 entradas analógicas. Adicionalmente incluye un puerto PLK, un puerto Ethernet y dos entradas USB. En este dispositivo se carga el programa desarrollado en el software de Automation Studio y será el encargado de la coordinación de todo el sistema.

La alimentación del PLC viene dada por dos entradas de uso específico, una de 25 voltios y otra a 0. Además, otras dos señales digitales de salida se han usado como señales de habilitación de ambos servoaccionamientos. El PLC se encuentra también conectado con un rúter por el puerto Ethernet, a través del cual intercambia datos con la App móvil, y con

ambos servoaccionamientos por el puerto PLK con los que envía y recibe datos de distintos parámetros y a los que envía las señales de control correspondientes.

6. Rúter wifi genérico.

Elemento bastante conocido por su presencia en el hogar el cual realiza una función similar dentro de este sistema. Habitualmente el rúter tiene como función establecer una red conectada a la red de telefonía que permite la conexión a la misma de distintos dispositivos. En este caso, también ejerce la función de crear una red para conectarnos a través del dispositivo móvil y poder realizar el manejo de los motores a través de una aplicación, pero en este caso no intercambia datos con la red de telefonía si no con el PLC.

Por tanto, este tiene una única entrada fija a través del puerto Ethernet que se conecta con el PLC. Pero también tiene una entrada de datos a través de dicha red wifi.

7. Servomotores síncronos 8LSA57.DA030S000-3.

Se puede definir servomotor como una mejora de los motores convencionales que permite el control de la posición, la velocidad y el par con gran precisión. Normalmente se encuentra compuesto por un motor eléctrico, un sistema de regulación, un encoder y un potenciómetro (para conocer la posición en la que se encuentra).

En este caso se trata de un motor eléctrico de la clase 8LS. Se trata de un motor síncrono de 4 polos, velocidad nominal de 3000 rpm y par resistente 17,5 Nm. Al tratarse de un servomotor, se tiene la posibilidad de manipular las condiciones de ambos motores para realizar distintas pruebas en distintos puntos de funcionamiento.

Dado el objetivo experimental que tiene este proyecto, los motores se encuentran situados frontalmente unidos por su eje central. De esta forma será posible experimentar con múltiples casos de par resistente y velocidad de giro y ver el comportamiento del sistema. Además, estos motores se encuentran conectados por 3 fases y un neutro con sus correspondientes servoaccionamientos.

Con la introducción realizada a cada uno de los elementos y su relación dentro del sistema ya se tiene una visión general del proyecto físico. Puede quedar como duda si no es necesario el dimensionamiento del cableado de este proyecto. La respuesta es no, las especificaciones de los cables usados vienen determinados por los manuales de cada uno de los elementos del sistema. Donde se especifican, tanto las características de los cables (material, sección, aislamiento y longitud) como las de las protecciones. Además de las medidas a considerar en la disposición de los elementos físicamente. Estas consideraciones se han seguido como aparecen indicadas y por tanto no tiene mayor interés su inclusión en esta memoria.

5. Programa AUTOMATION STUDIO.

5.1. Introducción

Automation Studio es un software creado por la empresa austríaca B&R. B&R es una de las empresas punteras en el entorno de la automatización industrial, cuyo pilar básico es la innovación dentro de dicho sector. Esta empresa forma parte del grupo multinacional ABB, al cual proporciona soluciones en el entorno de la automatización de máquinas y fábricas.

Desde B&R definen a Automation Studio como "el entorno de software integrado que contiene herramientas para todas las fases de un proyecto". Y así es, Automation Studio posibilita el control total del sistema, además de servir el mismo como plataforma de visualización.

Una de las ventajas de Automation Studio es la programación. Este aboga por una arquitectura modular con una programación sencilla y segura. Como puntos principales se puede destacar el uso de dependencias del hardware en uso y las librerías suministradas con Android Studio para facilitar la programación de módulos con funciones concretas. Como se verá más adelante dichas librerías específicas para distintas funciones facilitan en gran medida el trabajo realizado.

Otra de las ventajas a nivel de programación de Automation Studio es la cantidad de lenguajes que soporta, permitiendo al usuario elegir el que sea de su preferencia sin perder ninguna posible funcionalidad. Incluye los lenguajes de la norma IEC 51131-2 (LD, FDB, IL, SFC y ST), Automation basic, CFC y ANSI C; además de C++ con la instalación de un paquete adicional.

Como es oportuno, se ha seguido la norma general del ámbito de la programación de automatismos y el proyecto se basará en Structured Text, el lenguaje más usado dentro de los que conforman la norma. Structured Text (ST) o Texto estructurado se trata de un lenguaje de alto nivel como puede ser C. Su principal ventaja es que facilita la comprensión del código (sobre todo a terceros) en programas extensos donde lenguajes gráficos pueden hacerse complicados de leer.

5.2. Estructura del proyecto de Automation

Como se menciona en el apartado anterior, Automation Studio tiene un enfoque modular a cada una de las funcionalidades necesarias y así se ha pretendido enfocar. De esta forma, surgen tres módulos diferenciados: Motion 1, Motion 2 y TCP Server y un Programa principal que los coordina.

A continuación, se añade un esquema que muestra las relaciones entre estos módulos y el flujo de información.

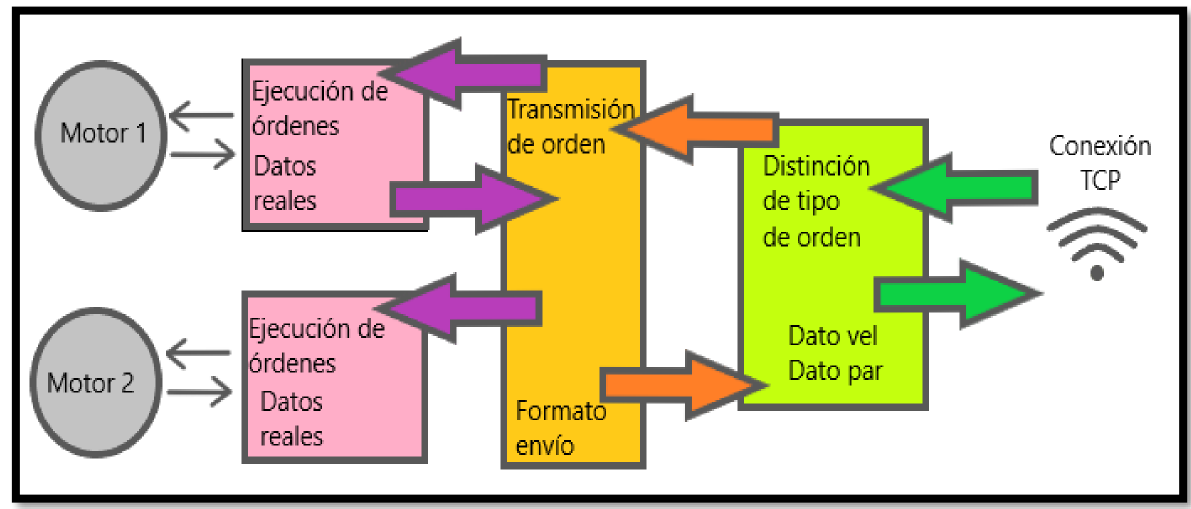


Ilustración 2. Comunicación entre los módulos del proyecto desarrollado en Automation Studio. Fuente propia. La imagen muestra esquemáticamente la relación entre los distintos módulos del proyecto y la función básica que ejecutan.

Cabe mencionar que el diagrama pretende representar de forma sencilla la relación entre módulos. Además, muestra la relación "virtual" que tienen con el exterior, con la que se representa el origen y destino de los datos en el exterior.

Del diagrama se pueden deducir dos flujos de datos principalmente, orden App-Motor y datos Motor-App. Se incluye una explicación resumida de cada una para aclarar la organización del código antes de su análisis.

5.2.1. Orden App-Motor

Cuando un usuario ejecuta una orden desde la aplicación, esta se envía en forma de número hexadecimal (Ver apartado 4.6. Códigos de comunicación) por protocolo TCP/IP y se recibe a través del Router que se encuentra conectado al PLC. El módulo encargado de interpretar y recibir esta entrada de datos es el que se ha llamado TCP Server.

El módulo TCP server es el encargado del establecimiento de un servidor TCP, de la gestión de conexión de dispositivos y de la recepción y envío de información. Este recibe la orden y la almacena de inmediato en una variable. Justo tras recibir la orden se distingue si esta se trata de una orden binaria (Pulsado de botón) o una orden entera (cambios de velocidad o par resistente). En cada caso se almacenan en variables globales distintas y se activa una variable binaria global que indica que se ha recibido dicho tipo de dato. Hecho esto, el proceso sigue desde el programa principal.

El programa principal funcionando en modo cíclico reacciona a las ordenes de tipo entero mencionadas en el párrafo anterior. En este se reacciona únicamente a los datos ya filtrados como enteros distinguiendo si se trata de un dato de velocidad o de par, tras esto se varían unas variables globales que aplican el cambio correspondiente en el módulo Motion. Además se encarga de activar y desactivar los bits de Enable de los motores en el caso de que la orden recibida sea la correspondiente a dicho caso.

El Motion también se encuentra funcionando cíclicamente y, al cambiar una de las variables globales, el programa dentro del Motion reaccionará para ejecutar la acción correspondiente. Es importante considerar que, el hecho de que la ejecución se lleve a cabo, o no, depende del estado de la estructura de órdenes que se encuentre el servomotor (Ver 5.5 Motion para más detalles). En resumen, dependiendo del estado de su estructura en el que se encuentre el Motion podrán suceder tres cosas: que se ignore la orden, que se ejecute la orden, o que se genere un error. Estas posibilidades también se describen en el apartado de Motion.

5.2.2. Datos Motor-App

Como se ha mencionado anteriormente y se seguirá haciendo referencia más adelante, la idea del proyecto no es sólo que se envíen instrucciones desde la App Android a los motores, sino que dicha App a su vez pueda recibir datos de estado de estos motores. Este proyecto se ha desarrollado para que este retorno de datos sea únicamente de la velocidad, aunque tanto los códigos de comunicación como el código fuente del programa se han desarrollado dejando hueco a la posible inclusión de retornos de datos del par.

El flujo de datos, en este caso, empieza en el encoder interno que porta el servomotor. Los datos de dicho encoder son recibidos por el módulo Motion 1, que los almacena de forma continua en una variable local. Esta variable local se carga en una global una vez por ciclo. Dicha variable global se transforma de las unidades del sistema a rpm, y se pasa al formato del código de comunicación en el módulo principal. Una vez en dicho formato se actualiza en el valor de una variable global que se encuentra enviándose continuamente por TCP.

5.3. Automation Studio – Preparación del proyecto

Una vez conocida la estructura del proyecto y habiéndose definido superficialmente el flujo de datos por el programa, se hace una pequeña introducción al software y se explica cómo se sientan las bases para comenzar a programar un proyecto como este. Al tratarse de un programa para una función muy específica, la comunidad y los recursos online no son tan extensos como los disponibles para otro tipo de software, como puede ser Android Studio. Por ello, se muestra detalladamente esta puesta en marcha y comienzo de uso del programa para que pueda servir de base para otros que pretendan trabajar con dicho software en futuros proyectos.

5.3.1. Introducción al software

En este apartado se pretende dar las nociones básicas de las distintas partes del software. Para ello se explica primero la vista por default que se muestra al ejecutar el programa.

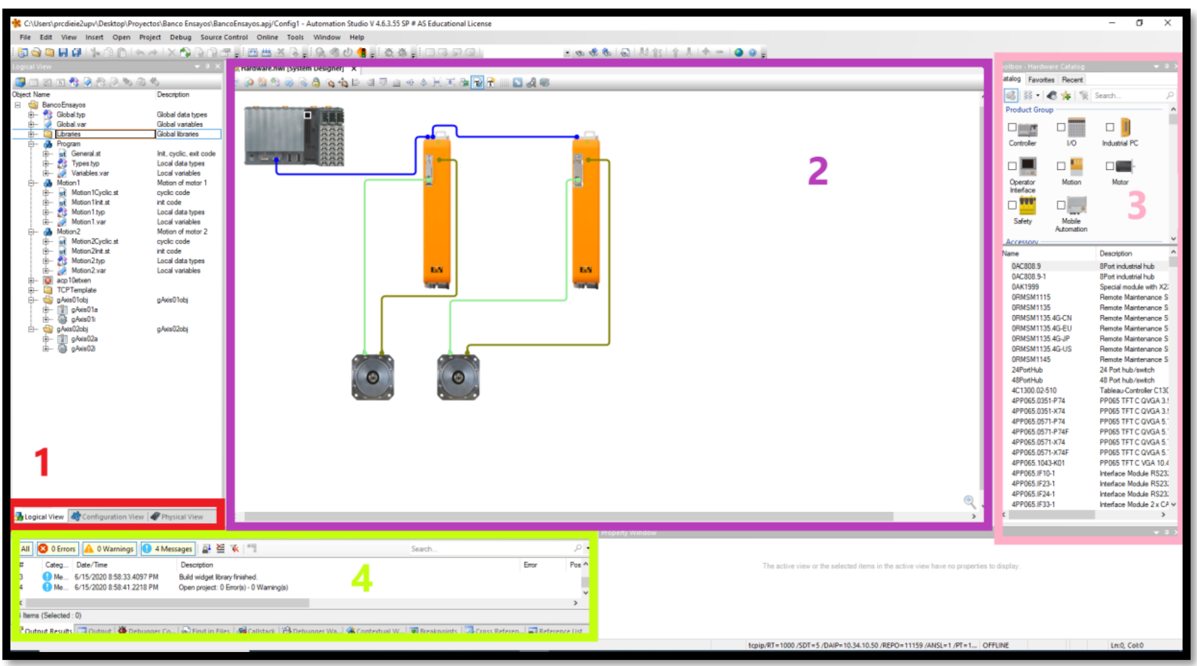


Ilustración 3. Captura general del programa de Automation Studio. Fuente propia. Se detallan las distintas partes en las que se divide el entorno de trabajo de este software y se describen a continuación.

La imagen superior muestra la vista del proyecto al ejecutarlo. Se han marcado y numerado las secciones más relevantes para el uso general y se comentan brevemente a continuación.

1. La sección marcada en rojo se trata de tres pestañas para acceder a las secciones: Logical View, Configuration View y Physical View. Al saltar entre ellas se muestra su contenido en la parte superior permitiendo acceder a los distintos campos. Estos tres apartados contienen el grueso del proyecto. A continuación, se da una pequeña introducción a cada uno de ellos mencionando sus funciones principales.

- Logical View. Contiene todo lo referente al código que se va a cargar en el PLC. Tanto los distintos módulos del programa como las librerías necesarias para ejecutarlos.

- Configuration View. Principalmente controla todos los detalles de funcionamiento del PLC, los distintos ciclos de ejecución de programa y toda la configuración de funcionalidades como motion o Mapping (no usado en este proyecto).

- Physical View. Controla la configuración de los distintos elementos de la vista, como elementos dentro del sistema. Es decir, no está tan dedicado al funcionamiento interno del programa (como está la configuration View), sino a la comunicación y relación entre los dispositivos y el exterior.

2. La sección marcada en lila encuadra dónde se ejecutarán las distintas ventanas de código, configuración y otras funcionalidades. En la imagen se muestra abierto el archivo hardware.hwl, el cuál será configurado más adelante.

3. En la sección 3 marcada en rosa, aparece por defecto el hardware catalog, que como se verá más adelante no sólo sirve para añadir elementos al System Designer, sino que también permite la búsqueda y selección de librerías adicionales y plantillas como la de Motion o servidor de TCP que van a ser usadas en el proyecto. Durante el uso del programa esta pantalla va mostrando distintas opciones en función de lo que se esté modificando.

4. Por último, la sección 4 se trata de la sección de Output Results, donde se mostrarán los resultados de la ejecución y posibles errores en el código. Esta ventana es común a distintos softwares de programación y permite el uso de funcionalidades típicas como la introducción de breakpoints o visualización y localización de errores.

5.3.2. Preparación de la Physical View

El primer paso es la inclusión de los elementos que contiene el sistema en la Physical View. Para ello, al crear un nuevo proyecto se da la opción de añadir un primer elemento, el cual es conveniente que sea el PLC que se está utilizando, en el caso de este proyecto, el X20CP1382. Una vez añadido el primer elemento los siguientes elementos se pueden añadir desde el Hardware catalog situado a la derecha de la pantalla.

Desde este Harware catalog se deben seleccionar dos servoaccionamientos del tipo ACOPOS P3, especificando el modelo específico usado. Antes de realizar este paso, es importante entrar en la página de producto de los servomotores y descargar la extensión para que aparezca disponible el modelo de servomotor que se va a utilizar, así se podrá añadir el servoaccionamiento al mismo tiempo. A continuación, se aporta una secuencia de imágenes que guía el proceso de forma adecuada.

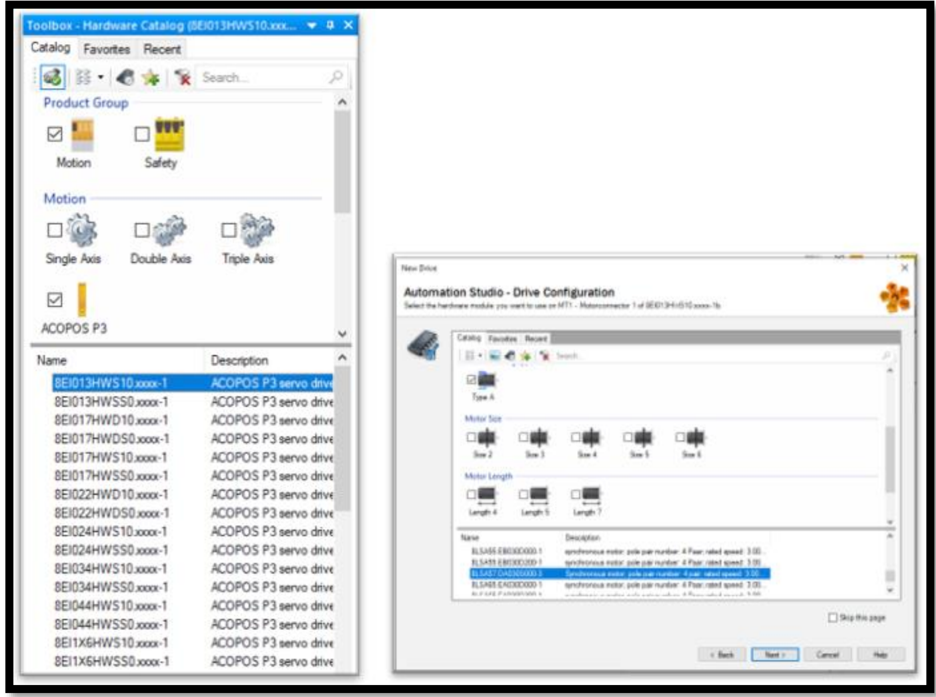


Ilustración 4. Capturas del Harware catalag y sección de selección de motores. Fuente propia.

Tras añadirlo aparecerá una ventana para seleccionar el encoder utilizado, donde no se debe realizar ninguna selección. El modelo usado usa el sistema EnDat, este permite una detección automática sin necesidad de especificarlo. Sólo habrá que marcar la casilla correspondiente que indique que el servoaccionamiento sigue este protocolo.

Tras esto, la siguiente ventana que se muestra es la ventana de selección del servomotor. Como se observa tiene almacenados de serie varios modelos, pero si el modelo que se está usando no aparece, se debe ir a la página del producto en B&R y descargar la extensión.

A partir de aquí, sólo queda seguir los pasos marcados en la "Guía básica de Motion ACP10" disponible en la página web y se habrán añadido los ejes satisfactoriamente. Se deben repetir los pasos tantas veces como motores a instalar. Una vez los ejes se hayan añadido, se deben realizar las conexiones de PowerLink por las que el PLC se comunica con los servoaccionamientos. En el apartado de montaje físico ya se encuentra descrito cómo se ha realizado dicha conexión. Uno de los servomotores llamado se encuentra conectado directamente con el PLC, y el otro se encuentra conectado al anterior (en caso de tener más dispositivos se colocan uno tras otro de la misma manera). El resultado en la vista Hardware.hwl [System Designer] es el mostrado en la siguiente imagen.

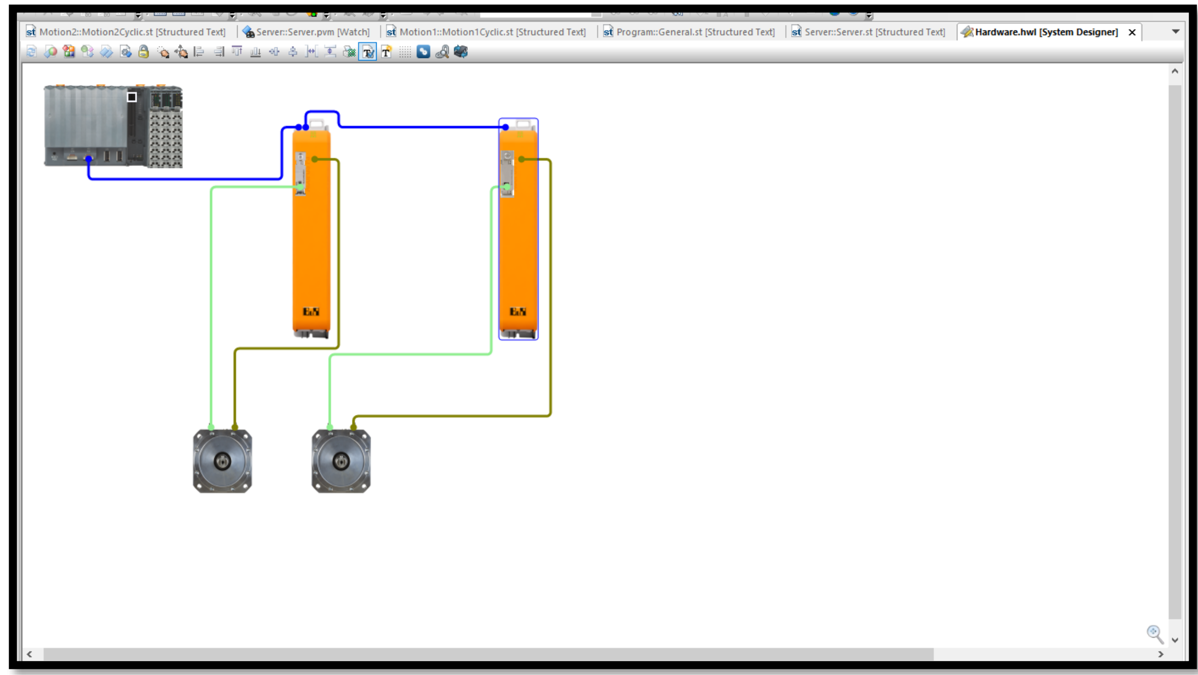


Ilustración 5.Vista del System Designer de Automation Studio. Fuente propia. En esta se visualizan el PLC, los dos servoaccionamientos y los motores conectados entre sí.

Como se puede ver, se muestra una representación visual de cómo queda organizado el sistema. La conexión entre los servomotores y servoaccionamientos aparece por defecto. Mientras que la conexión que simula la conexión PLK hay que realizarla de forma interactiva en la vista.

5.3.3. Dynamic Node Allocation

Como se ha mencionado, los servoaccionamientos ACOPOS P3 se comunican con el PLC mediante conexión POWERLINK. Para que esta conexión se lleve a cabo de forma satisfactoria cada equipo ACOPOS P3 presente en la red debe tener un número de nodo distinto del resto de equipos con los que comparte red. Tradicionalmente, los servoaccionamientos incluyen un conjunto de interruptores que permite indicar el número de nodo, sin embargo, este sistema no presenta esa funcionalidad. Los ACOPOS P3 permiten dos formas para configurar el número de nodo:

1. Configuración externa mediante una pantalla accesoria 8EAD.
2. Configuración por programa utilizando la función Dynamic Node Allocation (DNA) de POWERLINK.

Por no disponer de la pantalla accesoria, esta localización de nodos se deberá hacer a través de la función DNA. Este proceso se encuentra explicado en la guía básica detalladamente, a continuación, se muestra la configuración para el caso actual.

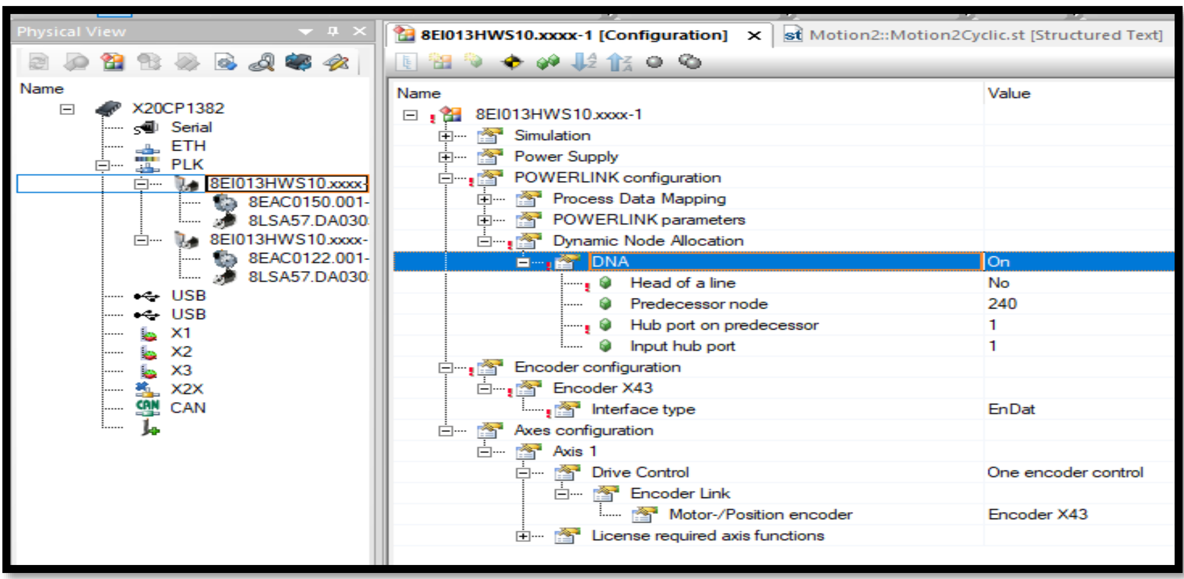


Ilustración 6. Captura de la sección de configuración del DNA. Fuente propia. En la imagen se muestra el apartado de configuración DNA con las características del primer motor de línea.

Como se indica en la guía de usuario para configurar el Dynamic Node Allocation de ambos motores, se debe acceder al apartado de physical view, seleccionar cada servomotor y entrar en el apartado de DNA. Configurando:

- Para el primer motor es la configuración que se muestra en pantalla. Se activa la opción DNA (Dynamic Node Allocation), se selecciona como no cabeza de línea ya que los ACOPOS P3 no pueden serlo a no ser que se configuren por pantalla accesoria. El nodo predecesor, al ser el primer servoaccionamiento, corresponde con el maestro de la red POWERLINK (el PLC). El Hub port on predecesor se debe dejar a ‘1’ por ser el primer esclavo de la línea. Y el Input hub port también debe estar configurado a ‘1’ ya que todos los ACOPOS P3 tienen como puerto de entrada de datos el ‘1’.

- Para el segundo motor la configuración es la misma excepto que el nodo del predecesor se debe poner a ‘1’ ya que el predecesor es el esclavo ‘1’, y el Hub port on predecesor debe ser ‘2’ ya que la salida del primer servomotor se realiza por el puerto POWERLINK 2.

Además, en esta misma pantalla debe marcarse la interfaz del encoder del tipo EnDat en ambos servomotores, de esta forma el sistema reconocerá dicha funcionalidad.

5.3.4. Configuración IP

Para el fin del proyecto, es importante poder controlar la IP que se pretende que tenga el PLC dentro de la red. Para ello, hay que realizar un par de pasos que persiguen este fin.

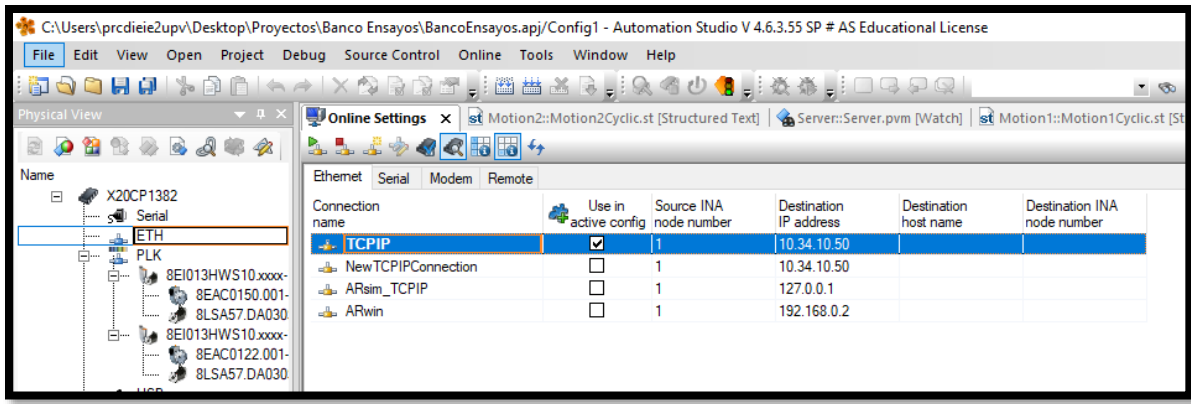


Ilustración 7. Captura de la sección Online settings de Automation Studio. Fuente propia. Se muestra la IP configurada junto el uso en configuración activa para forzar el uso de dicha IP.

Como se ve en la imagen hay que entrar en la physical View y entrar en la configuración online del puerto Ethernet. Aquí hay que cambiar la IP por la que queremos fijar al PLC y marcar la casilla de Use in active configuration. Con este cambio se está configurando la detección de dicho dispositivo a la hora de cargar el programa.

Además, se debe configurar en la sección de configuración del PLC, también accesible desde la Physical View, que el PLC (X20CP1382) tenga como parámetro interno la IP que se requiere y la submask.

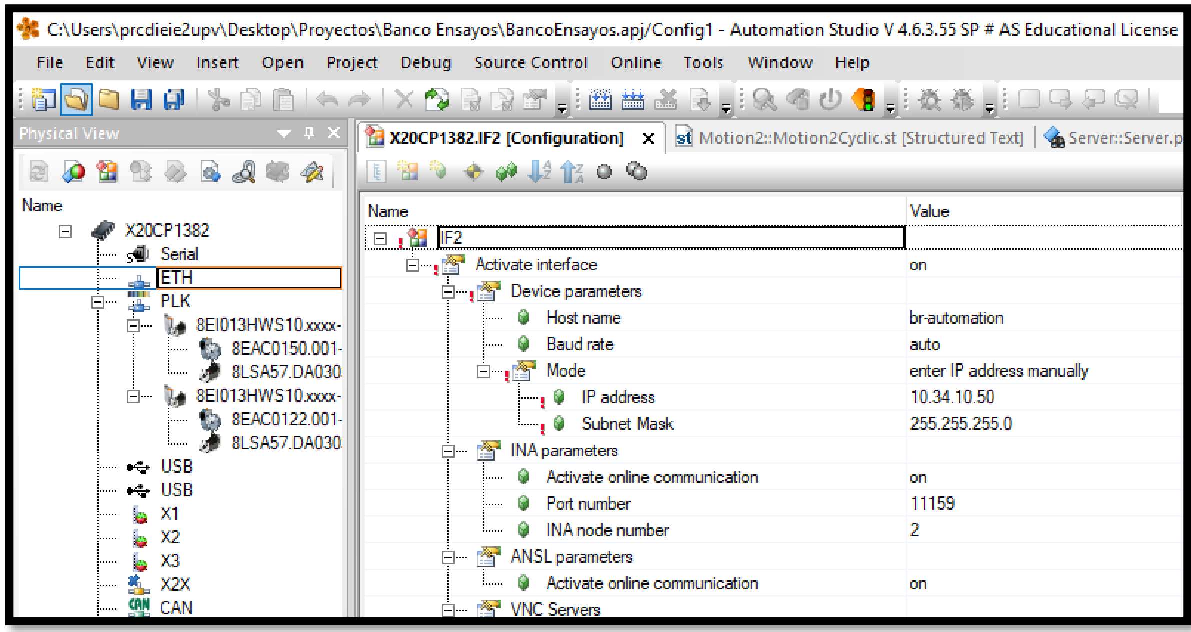


Ilustración 8. Captura del apartado de configuración del PLC. Fuente propia.

Respecto a este apartado, cabe mencionar que la IP ha sido seleccionada arbitrariamente pero teniendo en cuenta las convenciones típicas utilizadas por el servicio técnico de B&R y que podría ser cualquier otra cuyos primeros “campos” coincidiesen con la del router que se está usando para establecer la conexión a la red. Este router ha sido fijado al IP 10.34.10.51 y una submask, 255.255.255.0, también de forma arbitraria y atendiendo a las convenciones mencionadas.

5.3.5. Librerías

Las librerías suponen una parte importante a configurar. Estas permiten que numerosas funciones y tipos de variables puedan ser usados, facilitando la reducción de mucho tiempo de programación.

Como se ha mencionado en el apartado 5.3.1. Introducción al software, las librerías necesarias para llevar acabo módulos como el de Motion o TCP se encuentran en la sección de harware Catalog. Estas se deben añadir a la carpeta Libraries en la Logical View como se muestra en la imagen.

Normalmente, al añadir una librería nueva se añade automáticamente los Library Objects. Pero no siempre es así, en este caso se deben añadir manualmente en el archivo Cpu.sw (presente en la figura superior) que se encuentra en la Configuration View. Es importante tener este punto en cuenta ya que es una posible fuente de error que en ciertos casos no resulta evidente.

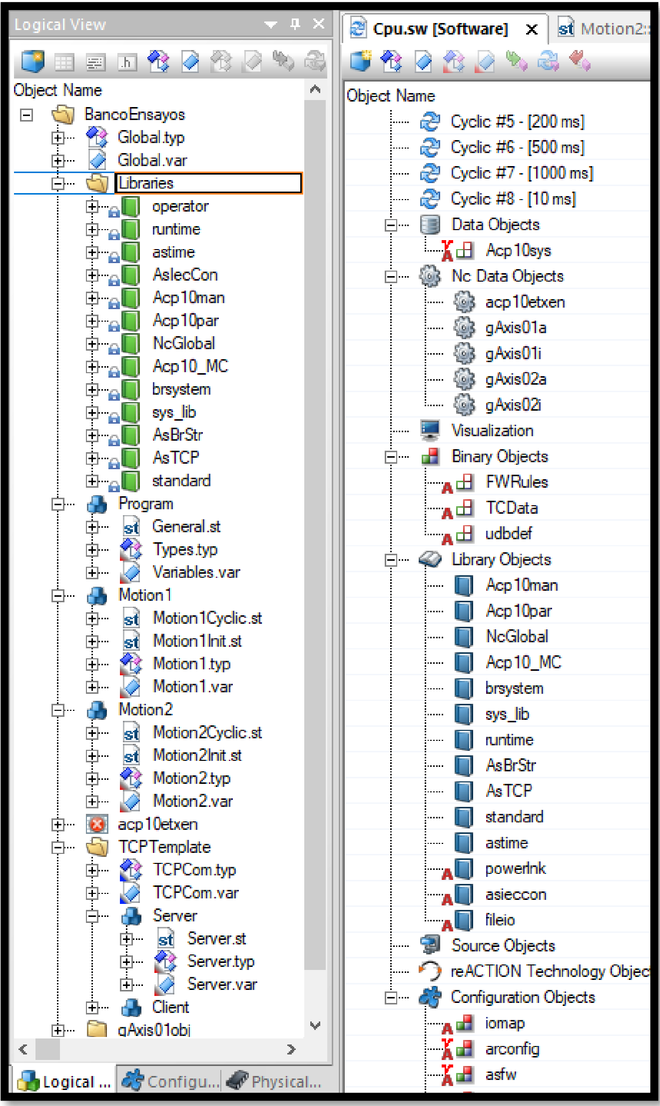


Ilustración 9. Captura del fichero Cpu.sw y carpeta de librerías. Fuente propia.

Automation Studio ya viene con algunas librerías por defecto para su funcionamiento normal, pero para este proyecto se han tenido que añadir algunas para desarrollar ciertas funciones específicas. Para el módulo de motion, se han añadido las librerías: Acp 10_MC, brsystem, Acp 10man, AsBrStr y Acp10par. Y para el módulo de TCP se ha incluido la librería de AsTCP.

5.3.6. Introducción de programas y velocidad de ciclo

Como se ha visto hasta ahora, en el PLC se encontrarán 4 programas funcionando a la vez que interactúan entre ellos. Para introducir dichos programas hay que seguir ciertas indicaciones. El resultado final se puede ver en la imagen anterior.

Un nuevo programa se puede añadir fácilmente una vez abierta la Logical View en el recuadro donde suele aparecer el hardware catalog, aparece un Object catalog, donde uno de los elementos aparece como “ST all in one”; arrastrando el mismo se crea la base del programa. Dicha base incluye

un fichero de código “General.st”, un fichero de tipos propios “Types.typ” y, por último, un fichero de variables locales “Variables.var”.

Para añadir los ficheros de motion se debe acceder al mismo Object Catalog y filtrar la búsqueda por programas. En este caso se han elegido las plantillas necesarias para un eje y se han arrastrado a la Logical View, quedando como se muestra en la imagen.

Por otro lado, la plantilla de comunicación TCP fue suministrada por B&R. Por lo que si no se encuentra en la página web es necesario contactar al servicio técnico.

Una vez se tienen los programas incluidos en la Logical View, se debe comprobar que estos se encuentran añadidos en el fichero Cpu.sw en el ciclo de funcionamiento adecuado.

Cpu.sw [Software] | Motion2::Motion2Cyclic.st [Structured Text] | Server::Server.pvm [Watch] | Motion1::Motion1Cyclic.st [Structured Text] | Program::General.st [Structure

Object Name	Version	Transfer To	Size (bytes)	Date	Source	Source File	Description
<CPU>							
Cyclic #1 - [10 ms]							
Server	1.00.0	UserROM	8032	3/12/20...	TCPTemplate.Server	Config1\X2...	TCP Server SAMPLE program
Cyclic #2 - [20 ms]							
Cyclic #3 - [50 ms]							
Cyclic #4 - [100 ms]							
Program	1.00.0	UserROM	3112	3/12/20...	Program	Config1\X2...	
Motion1	1.00.0	UserROM	25844	3/12/20...	Motion1	Config1\X2...	Motion of motor 1
Motion2	1.00.0	UserROM	24884	3/12/20...	Motion2	Config1\X2...	Motion of motor 2
Cyclic #5 - [200 ms]							
Cyclic #6 - [500 ms]							
Cyclic #7 - [1000 ms]							
Cyclic #8 - [10 ms]							

Ilustración 10. Captura del fichero Cpu.sw sección de ciclos de funcionamiento. Fuente propia. En esta se visualizan los distintos módulos funcionando en sus ciclos de funcionamiento marcados.

Dichos ciclos de funcionamiento controlan a qué velocidad se ejecuta el código cíclico de cada programa. En ciertos casos, este puede ser configurado en función de la reactividad que se quiera en ciertas funciones y en otros depende de las prestaciones de cada función.

En el caso del servidor TCP es función de este tipo de comunicación que requiere una velocidad de respuesta muy elevada. Por ello su programa cíclico se ejecuta cada 10 ms. Por otro lado, los programas Program, Motion 1 y Motion 2, se ejecutan en un ciclo de 100 ms. En el caso de los módulos de motion viene especificado en la guía. Este hecho puede estar asociado a la capacidad de procesado de los módulos de control ACOPOS. Por último, el programa sí se tiene libertad para ejecutarlo a una velocidad elegida, en este caso se ha colocado a 100 ms por ciclo.

5.3.6. Ajuste del controlador, primer arranque y errores en primera puesta en marcha

Realizar un ajuste de los parámetros del controlador es conveniente para optimizar la repuesta del mismo. Aunque esto se puede realizar manualmente, para conseguir la respuesta más dinámica posible se puede hacer uso de la función Autotuning. Para una guía sobre esta función diríjase a la Guía básica de motion.

Para realizar un primer arranque y comprobar que los módulos de motion funcionan correctamente no es necesario comenzar a programar código. Automation tiene una función para controlar los motores desde un panel, lo que permite realizar las primeras operaciones y comprobar que no hay ningún error que solucionar antes de comenzar a desarrollar la aplicación. A esta función de testeo se puede acceder haciendo click derecho sobre uno de los servomotores, en la vista harware.hwl, y seleccionar test, con lo que se abrirá una ventana que permite controlar el motor en su totalidad.

Tras realizar el ajuste del controlador y un primer arranque para comprobar que todo funciona correctamente se puede comenzar a trabajar en el código. En caso de que se de algún tipo de error se deben revisar los pasos mostrados anteriormente y recurrir a las guías de puesta en marcha para intentar solucionarlo. En esta fase del proyecto los errores pueden ser particularmente complicados de localizar ya que, a diferencia de cuando se trata de errores de código, la fuente del error no es indicada en el Error logger directamente. Por si se está en este caso, a continuación se listan unos posibles errores y cómo solucionarlos.

- Type data undefined. Aunque no se haya empezado a teclear código las plantillas de TCP y Motion ya hacen uso de variables y funciones que necesitan librerías adicionales. En caso de recibir el mensaje “Type data … undefined”, hay que comentar que se han añadido correctamente las librerías como se mencionan en el apartado 5.3.5. Librerías. En caso de que se encuentren incluidas las librearías necesarias es conveniente hacer un rebuild del proyecto, con esto se “limpian las bases” anteriores y vuelve a añadir algunas librerías que podían estar añadidas de forma errónea.

- PLK luz roja. En caso de cargar el proyecto y que en las luces en la parte superior de los ACOPOS se encuentre la del PLK en rojo, supone que hay un error en la conexión POWERLINK. El error principal en este caso es un fallo en la configuración de los nodos al hacer el DNA.

- Encoder Error. Por último, cabe mencionar que en caso de saltar un error de encoder, al ser el sistema EnDat, el error típico es el fallo al indicarlo, que se menciona en el mismo apartado de DNA (Dynamic Mode Allocation).

Tras esta explicación de cómo sentar las bases para comenzar un proyecto en Automation Studio, se debe comenzar con el desarrollo del código, lo que se tratará en los siguientes subapartados.

5.4. TCP server

Este programa se encarga de establecer un servidor TCP para habilitar su control desde un dispositivo remoto conectado a la misma red. Se encarga de registrar un dato cuando es recibido y de enviar continuamente el dato de velocidad del motor.

Para desarrollar este programa se ha partido de una plantilla de comunicación TCP que deja preparados, la gestión de la conexión, errores en la conexión y métodos de recepción y envío de datos. Ya que el código del funcionamiento general ha sido proporcionado por B&R, no es relevante

entrar en detalle de su funcionamiento, sólo se da una pequeña explicación de las distintas etapas del módulo y donde se ha desarrollado el código para recibir y enviar los datos deseados. Si se quiere profundizar más en el código se deja adjuntado en el anexo 2.

5.4.1. Inicialización y etapas

Antes de entrar en el programa cíclico se encuentra el programa de inicialización, en el caso del módulo TCP, en este se da la configuración del puerto en el que se aloja el servidor, y condiciones de funcionamiento.

Tras la configuración de las primeras etapas se entra en el programa cíclico, en este se plantea una estructura case que alterna entre distintos estados de la aplicación. En la siguiente imagen se muestra un esquema que muestra el funcionamiento normal de dicha aplicación.

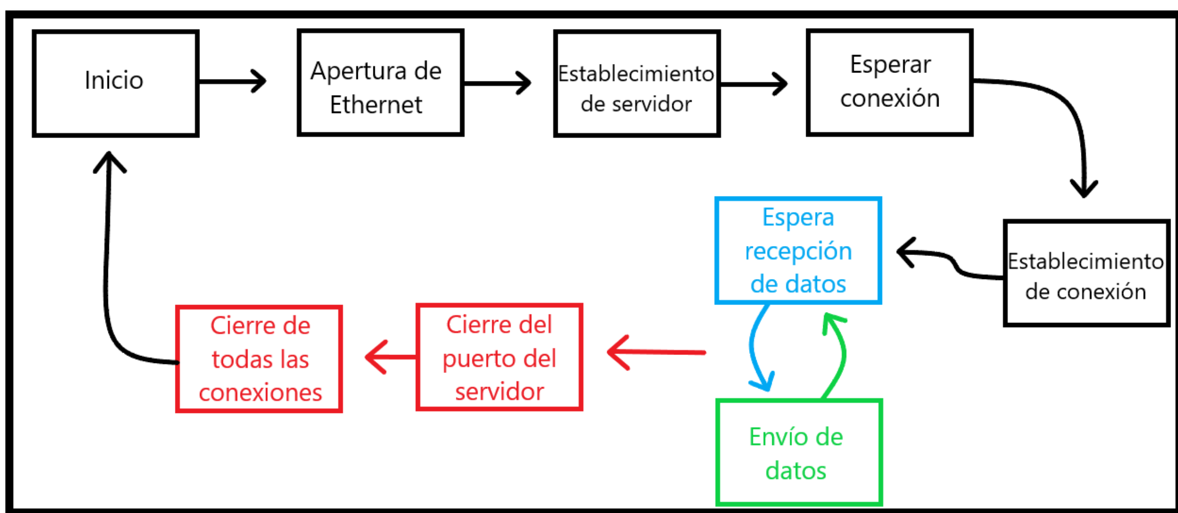


Ilustración 11. Esquema de funcionamiento de la estructura CASE del módulo TCP en Automation Studio. Fuente propia. En esta se muestran las etapas por las que pasa el módulo TCP para gestionar de forma correcta la conexión de dispositivos y el envío y recepción de datos.

En la imagen se muestra de forma simplificada las etapas por las que pasa el programa desde su inicio. La complejidad de código es mayor, pero con esta noción es suficiente para entender el uso que se le da en este proyecto.

La secuencia comienza en una fase de inicio que dura los primeros instantes hasta que se detecta la conexión ethernet. Cuando dicha conexión se da por completada, comienza un establecimiento del servidor. Una vez se establece el servidor se pasa a una etapa de espera, en la que se mantiene fijo hasta que un cliente requiere la conexión. Cuando se requiere la conexión comienza una etapa de establecimiento de la misma que termina con el inicio del temporizador de espera.

En el sistema anteriormente mencionado se hace de un uso de temporizadores (TON_Send_Server y TON_RecvTimeOut), que gestionan el tiempo máximo de envío y de escucha a la recepción de datos. Una vez iniciado uno de los temporizadores se comienza un ciclo en el que se pasa de fase si se acaba el tiempo máximo de envío o recepción, o se envía y se reciben datos. Este ciclo solo se

termina en caso de finalización de la conexión con el cliente. Este cierre de conexión se gestiona en dos etapas tras las que se vuelve a la fase inicial.

5.4.2. Recepción de datos.

Tanto este apartado como en el que viene a continuación, se centran en explicar específicamente el código utilizado para la gestión de los datos tanto de entrada como de salida, y no el funcionamiento total del esquema que ya incluía la plantilla para realizar dichas acciones.

Como es de esperar, el código para la gestión de los datos de entrada se incluye en la etapa de Espera recepción de datos mostrada en el gráfico. El código incluido es el siguiente:

```
ELSIF Server.TcpRecv_0.status = 0 THEN  (* Data received *)

        gServer.Status := 0; //COMMUNICATION OK
        (* Reset timeout timer, since data was received *)
        TON_RecvTimeout_Server.IN := 0;


        ResetValue := UDINT_TO_UINT(Server.data_buffer[0]);
        received := TRUE;

        IF ResetValue.15 THEN

                ReceivedData := (ResetValue AND 16#EF);
                DataChanged := 0;

        ELSIF NOT ResetValue.15 AND ActionDone THEN

                ReceivedAction := ResetValue;
                ActionDone :=FALSE;

        END_IF
END_IF
```

En esta parte específica, se hace una comprobación de si se han recibido datos, si este es el caso, se reinicia uno de los temporizadores mencionados anteriormente. En este punto en el que se tiene la certeza de que los datos han sido recibidos, dichos datos almacenados en la cadena Server.data.Buffer[], se almacenan en una variable. E inmediatamente se distingue de qué tipo de dato se trata en función del bit de mayor peso, almacenando el dato en variables distintas en función de su valor. La nomenclatura Variable.num_bit se usa para acceder específicamente al num_bit de dicha variable.

Cabe aclarar que sólo se almacena el primer elemento de la cadena porque se está trabajando con datos de bajo tamaño que se almacenan en una única variable. En caso de necesitar más espacio de almacenamiento habría que recurrir al uso de otras posiciones de dicho vector.

En el caso de tratarse de un dato de velocidad o par, este se almacena en una variable global ReceivedData filtrando el primer BIT que ya no tiene interés, además se pone la variable de DataChanged a 0 para notificar a otros módulos de que un dato nuevo ha sido recibido.

Un caso similar se da cuando se recibe un código de acción. Este código se almacena en una variable global y se notifica al resto de módulos marcando el Bit de acción realizada a 0.

5.4.3. Envío de datos.

El código desarrollado para el envío de datos se incluye dentro de la etapa Envío de datos mostrada en el gráfico. Cabe mencionar que realmente esta etapa se trata de dos etapas, una para el envío de datos y otra por si se da el caso de que el dato es de gran tamaño y no se puede realizar su envío en un único paquete. Sin embargo, como únicamente se están enviando datos de 16 bits, no se dará uso de esta segunda etapa.

El código para preparar el envío de datos es el siguiente:

```
IF alterner THEN

        SentData := TrueVelocity;
        (*Alterner := FALSE;
ELSE
        SentData := TrueTorque;
        Alterner := TRUE;*)

END_IF
```

Como se hará en el resto de apartados en temas relacionados con el envío de los datos reales del motor, se ha propuesto una posible estructura para el envío de datos de par contemplando la posibilidad de una expansión del proyecto en un futuro.

La idea básica del código realizado es almacenar la variable a enviar en una variable local SentData que en las siguientes líneas se incluye en los datos a enviar por TCP de la siguiente manera:

```
Server.TcpSend_0.pData := ADR(SentData);  (* Which data to send *)
Server.TcpSend_0.datalen := SIZEOF(SentData);  (* Lenght of data to send *)

END_IF
```

Dentro de la etapa se realizan muchas otras comprobaciones para asegurar su funcionamiento y gestionar posibles errores ocasionados, pero el código relevante a destacar es la inclusión de la variable SentData utilizada para almacenar los datos a enviar en los parámetros de datos a enviar y tamaño del paquete de envío.

Dichos datos se usan más adelante para gestionar el envío completo de dicho paquete de datos.

5.5. Motion modules

5.5.1. Motion General

En este apartado se tratan los puntos comunes de ambos módulos motion. Estos dos programas se encargan de permitir el control y recepción de datos de ambos motores. Ambos módulos comienzan de una misma base, pero sus programas serán modificados para ajustarse a las funcionalidades de cada uno.

Como se ha mencionado anteriormente, para desarrollar estos dos módulos se ha partido de una base que proporciona Automation Studio, que asegura un buen funcionamiento ante errores y órdenes, reduciendo los posibles peligros de un error en el proceso. A continuación, se añade un diagrama que muestra en qué se basa este pequeño esquema del que se parte, analizando las distintas fases del mismo.

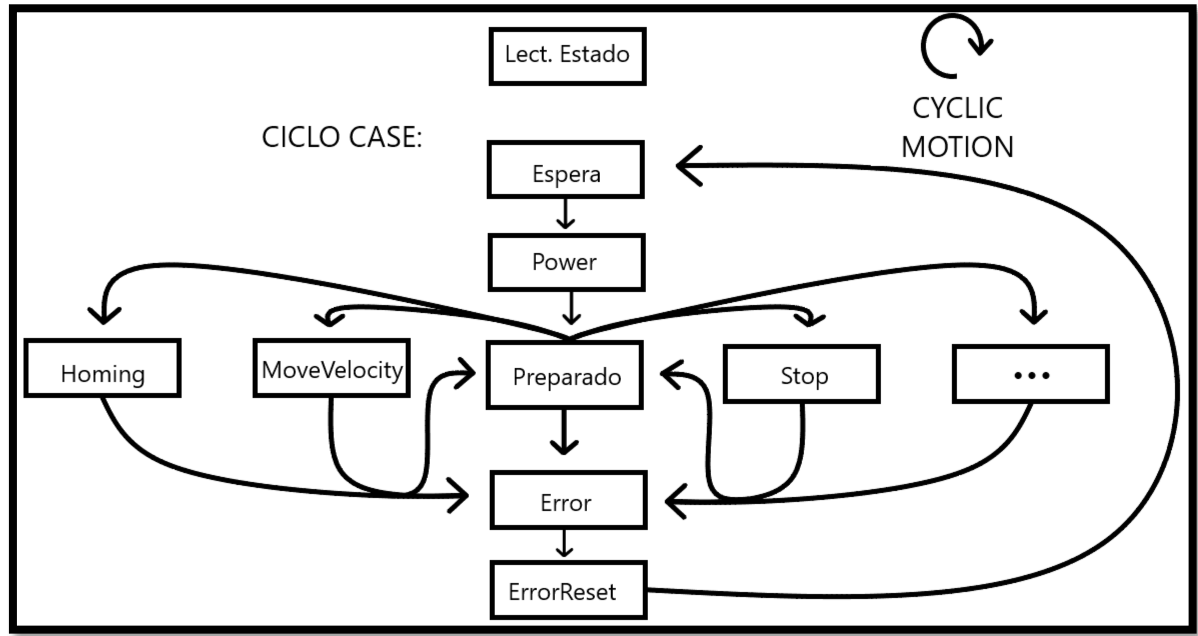


Ilustración 12. Esquema de funcionamiento de la estrucutra CASE en los módulos de Motion. Fuente propia. Ilustración similar a la ilustración 13 pero para la plantilla del módulo motion.

El esquema anterior no es una representación exacta de la organización del módulo, sin embargo, muestra el funcionamiento normal que tienen los módulos. Hay que resaltar que las flechas no indican que de una fase se vaya a otra inmediatamente. Esta estructura se da en forma de case (switch en ST) dentro de un código cíclico, las flechas representan una redirección en el próximo ciclo, es decir, si en el ciclo 1 se encuentra en el estado de preparado y se ejecuta una orden de paro, en el ciclo 2, se encontrará en el estado de Stop.

Cada recuadro representa una serie de instrucciones que realizan una acción "real" sobre el motor, por ejemplo, el estado MoveVelocity, se encuentran las órdenes necesarias para ejecutar un movimiento a una velocidad determinada.

Por tanto, en cada ciclo se realiza una lectura de Estado del motor, donde se comprueba el estado del eje, velocidad, posición y posibles errores entre otros. A continuación, se encuentra el ciclo case, que entra en el estado correspondiente.

El flujo normal en este ciclo es el siguiente: se comienza por una etapa de espera, una vez dada la orden de power, pasa al estado de power, se ejecutan las ordenes correspondientes y se pasa a un estado de Preparado. Este estado de preparado (o ready) se trata de otro estado de espera que aguarda una orden (Homing, Stop…) y cuando la detecta cambia de estado y, en caso de haberse realizado satisfactoriamente, vuelve a la etapa de preparado. En caso de detección de un error, se pasa a una fase de error que bloquea todos los movimientos. Para salir de dicho estado se debe recibir una orden de ErrorAcknowledge con lo que se pasa a resetear el error y a una fase de espera de nuevo.

Para interactuar con dicha estructura se da una estructura de variables llamada BasicControl para facilitar la ejecución de órdenes de forma segura. Esta consta de cuatro apartados: Command, Parameter, Status y AxisState. Modificando los distintos apartados de Command se puede seleccionar las actividades a ejecutar. En la sección Parameter se pueden modificar los parámetros con los que se quiere que se ejecuten ciertas acciones (p. ej., Cambiar la velocidad de giro). Status almacena los últimos datos leídos del eje. Y AxisState muestra el estado del eje (p. ej., Giro continuo, giro discreto, parada fija…).

A partir de esta base con ciertas adaptaciones, se ha realizado un código que gestiona las distintas órdenes que se requieren para cada motor. Cabe mencionar que el código completo de todos los módulos se encuentra en el anexo 2, en esta memoria solo se hará referencia a funciones puntuales para mostrar el funcionamiento del código.

5.5.2. Motion 1

Una vez aclaradas las distintas fases del ciclo por las que funciona el programa, cabe entrar al código que se ha desarrollado específicamente para este módulo. Se debe tener en cuenta que las funciones que debe desarrollar este módulo son: encendido, apagado, homing, arranque, freno y cambio de velocidad. Además, debe ser capaz de leer continuamente el dato de velocidad real del eje y enviarlo al programa principal para posteriormente enviarlo por TCP.

El código para realizar las funciones mencionadas se coloca tras la estructura case. Y hace uso continuo de la estructura de variables BasicControl. En este motion en particular no se modifica dicha estructura, sólo se hace uso de las funciones que proporciona.

5.5.2.1. Caso común

Se llama caso común a las órdenes que siguen una gestión similar. Estas órdenes se organizan de forma que cuando llega su código de orden correspondiente (6.2. Códigos de orden), se activa desde el BasicControl correspondiente para que se ejecute la orden y se marca una variable booleana para que no se vuelva a ejecutar ninguna acción hasta que se reciba otra por TCP.

En el siguiente control se muestra un ejemplo para realizar el Homing, que se extiende a los demás.

```
IF NOT ActionDone THEN

	CASE ReceivedAction OF

		7:

			BasicControl.Command.Home := TRUE;
			ActionDone := TRUE;

		9:
			...
	END_CASE

END_IF
```

Como se puede observar, primero filtra por la variable booleana ActionDone para conocer si ya se ha realizado la acción actual. En caso de no haber realizado la acción actual la estructura case lee la variable ReceivedAction que contiene el código de la acción. En el caso común, este ejecuta desde el BasicControl.Command la acción a ejecutar. En el ejemplo se ve el caso para el homing, este tiene un código binario equivalente a 7. Por último, marca que la acción se ha realizado.

Este caso común se repite para las acciones de ErrorAcknoledge, Run, Stop y Homing. En este mismo case también se ejecuta la acción de Power, pero esta se trata de un caso distinto.

5.5.2.2. Caso Power

Este caso de la estructura case se debe tener en cuenta que el mismo botón está pensado para activar y desactivar la entrada de potencia. Para ello, en este caso se plantea el siguiente código.

```
5:

	IF MC_Power_0.Status THEN
		BasicControl.Command.Power := TRUE;
	ELSE
		BasicControl.Command.PowerOff := TRUE;
	END_IF

	ActionDone := TRUE;
```

En este fragmento de código se muestra que en el caso de recibir una señal del botón Power, se debe consultar el estado en el que se encuentra el power del motor. En caso de estar activo, se usa el comando powerOff correspondiente para desactivarlo, mientras que si está inactivo se debe llamar al comando Power para que se desactive.

El comando PowerOff usado no está contemplado en la estructura base de Motion, por ello se debe añadir para poder usar dicha funcionalidad. Primero se añade un nuevo apartado en el BasicControl.Command para añadir la creación de una variable PowerOff. Para ello se tiene que ir

al listado de tipos locales y añadir una variable booleana PowerOff como se ve en la imagen siguiente.

Motion1::Motion1.typ [Data Type Declaration]* | Motion1::Motion1.var [Variable Declaration] | Motion1::Motion1Cyclic.st [Structured Text]* | Server::Ser

Name	Type	Reference	Replicable	Value	Description [1]
basic_command_typ			☑		command structure
Power	BOOL	☐	☑		switch on the controller
Home	BOOL	☐	☑		reference the axis
MoveAbsolute	BOOL	☐	☑		move to an defined position
MoveAdditive	BOOL	☐	☑		move a defiened distance
MoveVelocity	BOOL	☐	☑		start a movement with a defiened velocity
Halt	BOOL	☐	☑		stop every active movement
Stop	BOOL	☐	☑		stop every active movement as long as is set
MoveJogPos	BOOL	☐	☑		move in positive direction as long as is set
MoveJogNeg	BOOL	☐	☑		move in negative direction as long as is set
ErrorAcknowledge	BOOL	☐	☑		reset active errors
PowerOff	BOOL	☐	☑		switch off the controller
basic_parameter_typ			☑		parameter structure
basic_status_typ			☑		status structure
basic_axisState_typ			☑		axis state structure
basic_typ			☑		control structure

Ilustración 13. Fichero de declaración de nuevos tipos. Fuente propia. En la imagen se muestra el fichero Motion1.typ que permite la declaración de nuevos tipos de variables.

De esta forma, al declarar una variable BasicControl tipo basic_typ (que consta de cuatro variables de tipos: basic_command_typ, basic_parameter_typ, basic_status_typ y basic_axisState_typ), se va a tener acceso a una variable boolean PowerOff que se usa a continuación para deshabilitar el Power.

A continuación, se crea una constante STATE_POWEROFF que represente una etapa más de la estructura case. Dentro de esta etapa se debe realizar la desactivación del power. Esto se muestra en el siguiente código.

```
(******************** POWER OFF **********************)

STATE_POWER_OFF:
        MC_Power_0.Enable := FALSE;
        IF (MC_Power_0.Status = FALSE) THEN
                AxisStep := STATE_WAIT;//Si se ha apagado el motor vuelve a la fase wait
        END_IF
        (* gestiona el error *)
        IF (MC_Power_0.Error = TRUE) THEN
                BasicControl.Status.ErrorID := MC_Power_0.ErrorID;
                AxisStep := STATE_ERROR;
        END_IF
```

En la imagen superior se muestran las acciones necesarias para gestionar correctamente una desconexión del power. Cabe mencionar que la variable MC_Power_0 se ha declarado del tipo MC_POWER. Este tipo se trata de un bloque de funciones que requiere de una asociación al eje a controlar, en este caso el 1. Esto se realiza más adelante en el código (ver anexo 2).

Por último, es necesario configurar en el estado preparado que al diferenciar que la variable BasicControl.Command.Power se ha cambiado a '1' salte al estado correspondiente. Para ello se añade en la estructura IF que gestiona estas variaciones como se muestra a continuación.

```
(******************* READY *********************)

STATE_READY:  (* STATE: Waiting for commands ->AxisStep=10 *)

        IF (BasicControl.Command.Home = TRUE)THEN
                BasicControl.Command.Home := FALSE;
                AxisStep := STATE_HOME;

        ELSIF

                ...


        ELSIF (BasicControl.Command.PowerOff := TRUE) THEN
                BasicControl.Command.PowerOff := FALSE;
                AxisStep := STATE_POWER_OFF;
        END_IF

        END_IF
```

Como se puede observar la etapa STATE_READY consta de una estructura de IF e ELSIF para cada uno de los comandos. En este caso, se ha añadido un ELSIF que en caso de que .PowerOff se encuentre activo, borra la señal y va a la etapa STATE_POWER_OFF creada anteriormente.

Con los cambios realizados la funcionalidad PowerOff ha sido implementada satisfactoriamente.

5.5.2.3. Cambio de Velocidad

El cambio de velocidad se gestiona de una forma distinta. A este módulo llega el dato de la velocidad adaptado a las unidades del sistema. En este caso habrá que introducir la velocidad en BasicCommand.Parameter de la siguiente forma:

```
IF NOT DataChangedV THEN


        BasicControl.Parameter.Velocity := NuevaVelocidad;
        BasicControl.Command.MoveVelocity := TRUE;

        DataChangedV := TRUE;

END_IF
```

De esta forma se está ordenando que se ejecute el comando MoveVelocity, que equivale a Play, y al ejecutarse el mismo cogerá el parámetro de velocidad de NuevaVelocidad. Además, esto se hará únicamente cuando se halla recibido un nuevo dato. De esta forma no se satura la ejecución continua de una misma acción.

5.5.2.4. Velocidad de eje

De este eje únicamente falta por extraer su velocidad real. Para ello, añade una línea que almacena el dato en una variable global al ser leída en la plantilla. Esto se muestra en el siguiente código:

```
(******************* MC_READACTUALVELOCITY ********************)

MC_ReadActualVelocity_0.Enable := (NOT(MC_ReadActualVelocity_0.Error));
MC_ReadActualVelocity_0.Axis := Axis1Obj;
MC_ReadActualVelocity_0();
IF(MC_ReadActualVelocity_0.Valid = TRUE)THEN
	BasicControl.Status.ActVelocity := MC_ReadActualVelocity_0.Velocity;
	ActualVelocity := BasicControl.Status.ActVelocity;
END_IF
```

Este es el proceso que propone la plantilla para leer la velocidad del eje. Primero habilita la función, después la asigna a un eje y la inicia. Entonces, si la lectura es válida, lee la Velocidad actual y la almacena en los datos de Status de la variable BasicControl. Tras este almacenamiento se ha añadido una línea que almacena los datos recién leídos del eje a una variable global ActualVelocity.

5.5.3. Motion 2

En este proyecto el motor 2 tiene que aplicar un par resistente al motor 1. Para ello este módulo tiene que poder realizar las siguientes funciones: encendido, apagado, homing, errorAcknoledge y variación del par resistente. En este caso no se tiene que enviar ningún dato de vuelta a la aplicación.

Las cuatro primeras funciones se realizan con una estructura case de la misma forma que para el motion 1, la única variación consiste en los códigos de orden que se corresponden con las órdenes dirigidas al motor 2. Además, habrá que hacer la adaptación de la plantilla para incluir la función de PowerOff de la misma manera que en el apartado anterior.

Para el caso de la aplicación de par es necesario realizar un proceso similar al que se ha realizado para gestionar la funcionalidad de powerOff ya que la plantilla de motion no incluye esta funcionalidad.

5.5.3.1. Aplicación de par

Como se ha mencionado la aplicación de par no está contemplada en la plantilla de motion utilizada. Por lo que primero hay que crear una variable del tipo MC_TorqueControl y coordinarla con el eje. Ya que no se ha realizado anteriormente se enseña como se muestra una variable de este tipo con el eje en cuestión:

```
(************************ MC_TORQUECONTROL ***************************)

MC_TorqueControl_0.Axis := Axis2Obj;
MC_TorqueControl_0();
```

La primera línea asigna la variable de entrada del bloque de funciones axis con el eje número 2. Con la segunda se hace entrar en funcionamiento a dicho bloque. Ahora se debe añadir un nuevo paso a la estructura de case del programa que se llamará STATE_TORQUE, dentro de este se debe realizar una aplicación del par al motor 2,

```
(******************* APPLY TORQUE *********************)

STATE_TORQUE:

        // Resto de parámetros por ver
        MC_TorqueControl_0.Execute := TRUE;
        MC_TorqueControl_0.Torque := BasicControl.Parameter.Torque
        MC_TorqueControl_0.Acceleration:= 0;
        MC_TorqueControl_0.TorqueRamp := 0.5;
        MC_TorqueControl_0.Velocity := 20000;


        // Viendo si el par requerido se ha logrado ya.
        IF MC_TorqueControl_0.InTorque THEN
                MC_MoveVelocity_0.Execute := FALSE;
                AxisStep := STATE_READY;
        END_IF
        // Confirmar si se produce error
        IF MC_TorqueControl_0.Error THEN
                BasicControl.Status.ErrorID := MC_TorqueControl_0.ErrorID;
                MC_TorqueControl_0.Execute := FALSE;
                AxisStep := STATE_ERROR;
        END_IF
        MC_TorqueControl_0();
```

Primero se realiza una configuración de todas las entradas del bloque de funciones. Se ha seguido las recomendaciones estándar de la guía. Estas únicamente varían la aplicación del par en los extremos (a partir de la velocidad seleccionada) donde se reducen en función de la pendiente dada.

En el primer IF se comprueba si el par requerido se ha alcanzado ya para retornar a la fase STATE_READY. El segundo IF notifica en caso de que se haya realizado algún error durante el proceso de aplicación de par.

Una vez programada la etapa hay que añadir a la etapa STATE_READY en la estructura de IF una reacción a la activación de BasicControl.Command.Torque. A continuación, se añade al BasicControl.Command un apartado para realizar la aplicación de par y en BasicControl.Parameter una variable REAL Torque (que ya se ha usado en el código superior). Al ser muy parecido a las acciones mostradas en el apartado Motion 1 se anima al lector a revisar el anexo 2 para ver el código completo.

Por último, queda añadir tras el ciclo case el programa que reaccione a un cambio de par en la aplicación. Este es similar al cambio de velocidad mostrado:

```
IF NOT DataChangedT THEN

                BasicControl.Parameter.Torque := NuevoPar;
                BasicControl.Command.Torque := TRUE;
                DataChangedT := 1;

        END_IF
```

En caso de un nuevo dato de par recibido se almacena en el apartado de parámetros Torque y se activa el comando de ejecutar el par dado. Por último, se marca la realización de dicha acción.

5.5.3.2. Lectura de par

Aunque en este proyecto no se va a hacer uso de esta función, se ha implementado el código necesario para la lectura de esta para facilitar posibles expansiones. A continuación, se explica brevemente los cambios necesitados. Como la plantilla tampoco contempla la necesidad de una lectura de par, hay que comenzar añadiendo una variable del tipo MC_ReadActualTorque como se ha hecho con otras anteriores.

Para la lectura de dicho par se ha realizado el siguiente código, imitando la estructura vista para la lectura de la velocidad:

```
(******************** MC_READACTUALTORQUE *********************)

MC_ReadActualTorque_0.Enable := (NOT(MC_ReadActualTorque_0.Error));
MC_ReadActualTorque_0.Axis :=  Axis2Obj;
MC_ReadActualTorque_0();

IF(MC_ReadActualTorque_0.Valid = TRUE) THEN
                BasicControl.Status.ActTorque := MC_ReadActualTorque_0.Torque;
                ActualTorque := BasicControl.Status.ActTorque;
END_IF
        END_IF
```

Se empieza habilitando el comienzo de la lectura siempre que no exista un error en dicha función. Se sincroniza con el eje y si la lectura del par es válida se almacena en la estructura de BasicControl.Status (a la que hay que añadir una variable ActTorque). Con la intención de poder devolver dicho parámetro, se almacena en una variable global ActualTorque que se adaptará posteriormente en el MainProgram para su envío.

5.6. Main Program

Como se ha mencionado anteriormente, este programa sirve de gestor de algunas órdenes recibidas por PLC y de la adaptación de datos para ser enviados por el mismo sistema por TCP. Se gestionan las señales de habilitación de ambos servomotores Enable 1 y Enable 2. Se adaptan los datos de los códigos de comunicación propuestos a las unidades del sistema y viceversa.

5.6.1. Señales de habilitación

A nivel de código se trata de dos estructuras IF, una para la habilitación de las variables y otra para su deshabilitación. A continuación, se muestra el caso de la habilitación de variables:

```
IF ReceivedAction = 1 THEN
        Enable1:=TRUE;
ELSIF ReceivedAction = 2 THEN
        Enable2:=TRUE;
END_IF
```

Se analiza la acción recibida y en caso de coincidir con las que tienen función de habilitación se activa la variable correspondiente. La diferencia con las estructuras case vistas en los apartados de motion es que aquí no se depende de la variable ActionDone, ya que al tratarse de un comando de seguridad interesa que se ejecute sea cual sea el caso. Las excepciones que podrían llevar a la no ejecución de un Enable o Disable en caso de depender de ActionDone se detallan en el subapartado 4.6.1.1. Enable, Disable y ActionDone.

Las variables booleanas no tienen ningún efecto por sí solas sobre las señales de habilitación, para que se traduzcan en una salida del PLC hay que asignarlas. Para ello se ha accedido a la Physical View / X3 / I/O Mapping:

Channel Name	Process Variable	Data Type	Task Class
DigitalOutput01		BOOL	
DigitalOutput02		BOOL	
DigitalOutput03	::Program:Enable1	BOOL	Automatic
DigitalOutput04	::Program:Enable2	BOOL	Automatic
DigitalOutput05		BOOL	
DigitalOutput06		BOOL	
DigitalOutput07		BOOL	
DigitalOutput08		BOOL	
DigitalOutput09		BOOL	

Ilustración 14.Captura de la sección I/O mapping. Fuente propia. Permite seleccionar variables a salidas del PLC de forma que reflejen su estado.

En este hay que enlazar las salidas digitales deseadas con el valor de las variables Enable 1 y 2. De esta forma si toman el valor '1' se dará un voltaje de 24V y en caso de ser '0' se da tierra.

Cabe mencionar que se ha seleccionado la tercera hilera de salidas del PLC (X3), se pueden elegir salidas de X1 o X2 siempre y cuando se trate de salidas digitales.

5.6.1.1. Enable, Disable y ActionDone.

Como se ha mencionado en la parte superior, mientras que para ciertas órdenes se usa una variable ActionDone para determinar si la acción recibida ha sido realizada ya o no y ejecutarla en función de eso. Para las variables relacionadas con la señal de habilitación no se ha seguido este mismo esquema. Esto se debe a dos motivos:

1. Evitar que no se ejecute la orden por marcado de acción realizada.
2. Problemas ocasionados al realizar varias veces la misma acción.

En el primer caso se tiene en cuenta por la importancia que tienen dichos comandos. Cuando se quiere deshabilitar la señal de habilitación de un servomotor puede ser debido a su secuencia de acabado, pero también puede ser debido a que haya surgido una emergencia externa. En el caso de se trate de una emergencia se busca principalmente un factor a optimizar. Que la ejecución de la acción sea certera. Hacer dependiente su ejecución de ActionDone puede llevar a casos puntuales donde esta certeza se vea perjudicada. Véase el siguiente código.

```
IF NOT ActionDone THEN

    CASE ReceivedAction OF
        5:
            (************************** MC_POWER ****************************)

            IF MC_Power_0.Status THEN
                BasicControl.Command.Power := FALSE;
            ELSE
                BasicControl.Command.Power := TRUE;
            END_IF

            ActionDone := TRUE;
```

Ilustración 15. Captura del código de gestión de acciones del Motion 1. Fuente propia. Marca el intervalo donde en caso de producirse un cambio en la variable ActionDone se podría producir un error.

En la imagen se muestra el comienzo de la estructura case de la clase motion 1. Hay que tener en cuenta que las órdenes Enable Y Disable "conviven" con estas dos estructuras. Se usa el ejemplo del código anterior para explicar el problema pero pasa con todos los elementos de ambas estructuras case. Una vez que se ha entrado en el case y entrado en la orden correspondiente, hay un espacio de tiempo (marcado por el corchete rojo) donde en caso de cambiar de acción, el código terminaría con ActionDone verdadera y una acción no realizada.

Es decir, en caso de que se ejecutase una acción Power (como la de la figura) y en el espacio de tiempo marcado con el corche en rojo se quisiese ejecutar Disable, si la ejecución de disable dependiese del estado de ActionDone esta no se ejecutaría, disminuyendo su certeza.

Por otro lado, el motivo dos, que justifica esta diferencia es el tipo de acción que ejecutan. Todos los elementos que ejecutan acciones sobre el sistema del servoaccionamiento deben ejecutarse una sola vez ya que cada vez que se ejecutan generan una serie de acciones para poner dicha acción en funcionamiento. Por esto esta justificada la existencia de la dependencia de si la acción se ha realizado o no. Sin embargo, en el caso de las órdenes de enable y disable actúan sobre una salida lógica del PLC por lo que no generan ningún problema asociado si estas se ejecutan continuamente.

El caso ideal es que las acciones se ejecutasen repetidamente hasta la entrada de una nueva acción así se aseguraría que la acción se está realizando. Sin embargo, si esto se hiciese sobre acciones del motor se generarían continuos errores.

5.6.2. Adaptación de par y velocidad de entrada

El Main Program hace de intermediario entre el módulo TCP y el de los motores. En este se distingue entre dato de par y velocidad y se adapta a las unidades del sistema. Como se ha mencionado previamente, se recibe el tanto por ciento del dato. Por tanto, para aplicarlo al dato únicamente hay que multiplicarlo por el máximo del dato expresado en unidades del sistema y dividirlo por cien. A continuación se muestra el código encargado de esto.

```
LReceivedData := ReceivedData;

IF NOT DataChanged THEN

        DataChanged := 1;

        IF LReceivedData.14 THEN

                NuevoPar := PAR_MAX*(LReceivedData AND 16#DF)/100;
                DataChangedT := 0;

        ELSIF NOT LReceivedData.14 THEN

                NuevaVelocidad := VEL_MAX*(LReceivedData AND 16#EF)/100;
                DataChangedV := 0;

        END_IF
END_IF
```

Primero se actualiza la variable global con los datos enteros recibidos (ya mencionada en el apartado de TCP), y se guarda en una variable global con la que se trabaja de aquí en adelante. Este cambio de variable se encuentra justificado en el subapartado 5.6.2.1. LReceivedData , donde se destacan los errores que se evitan con esa medida.

Posteriormente, en caso de que se haya dado un cambio en el dato se marca la variable DataChanged como positiva y se distingue en función del valor LReceivedData.14. Como se ha

mencionado anteriormente, esta nomenclatura hace referencia al bit 14 del dato. Si se ve el apartado 7.3. Códigos de datos, se observa que se trata del bit de distinción entre dato de velocidad y par.

Una vez distinguido entre par y velocidad se adapta el dato como se ha mencionado anteriormente, con el uso del valor máximo de par en unidades del sistema. Además, se limpian los dos primeros bits de las variables ya que ya no tienen ningún uso. Tras esto se almacena el dato en una variable global y se pone la variable correspondiente de dato cambiado para "notificar" del cambio a los módulos de motion.

5.6.2.1. LReceivedData

El objetivo de este subapartado es discutir la necesidad de una variable local para almacenar la variable local que recibe los datos de par y velocidad. Al igual que en el apartado anterior se va a hacer referencia al código para ejemplificar el porqué de su necesidad.

```
IF NOT DataChanged THEN

    DataChanged := 1;

    IF LReceivedData.14 THEN

        NuevoPar := PAR_MAX*(LReceivedData AND 16#DF)/100;
        DataChangedT := 0;

    ELSIF NOT LReceivedData.14 THEN

        NuevaVelocidad := VEL_MAX*(LReceivedData AND 16#EF)/100;
        DataChangedV := 0;

    END_IF

END_IF
```

Ilustración 16. Captura del código de gestión de datos de velocidad y par del MainProgram en Automation Studio. Fuente propia.

Aunque las probabilidades de este hecho son reducidas, se puede dar el caso en el tras haber discernido si se trata de par o velocidad, el dato cambie en el pequeño espacio entre ambos. En dicho caso se le estaría aplicando un dato de par a la velocidad o viceversa. Para evitar dicha situación se usa la variable LReceivedData que deja fijo dicho dato mientras se opera con él.

5.6.3. Adaptación de datos de salida

Igual que en el apartado anterior se han adaptado los datos de entrada a las unidades de los motores, se deben adaptar los datos leídos de los motores para poder enviarlos de forma adecuada. Esto se gestiona en el siguiente código:

```
TrueVelocity := TRUNC(ActualVelocity*VEL_CONV);

TrueTorque := TRUNC(ActualTorque*TORQ_CONV) OR 16#80;
```

En este caso interesa actualizar lo más rápido posible los datos a enviar, por ello no se da ninguna condicionalidad. De esta forma al recibir un nuevo dato del estado del motor se adapta al formato de envío y ya se encuentra disponible para su envió por TCP.

Como se puede observar el dato de envío de velocidad se halla multiplicando por un factor de conversión, dicho factor se halla dividiendo la velocidad máxima en rpm entre la máxima expresada en unidades del sistema. Esto se aplica de forma similar para hallar el par, con la diferencia de que en este caso se añade un ‘1’ en el bit de mayor peso (Código de datos).

6. APLICACIÓN ANDROID

6.1. Introducción a Android

6.1.1. Android

Android es un sistema operativo basado en Kernel Linux, precisamente por estar basado en Linux está dotado de un núcleo de sistema operativo libre, gratuito y multiplataforma. Desde hace años es el principal sistema operativo en dispositivos móviles a nivel mundial, con una cuota de mercado entorno al 80%.

Hasta 2005 Android era un sistema operativo de muy poco renombre, en ese año fue comprado por Google, que tras dos años en su desarrollo sacó a la luz su primera versión, junto a una plataforma SDK para los desarrolladores. Durante los siguientes años Google fue mejorando y adaptándose a distintos dispositivos, llegando a la última versión lanzada recientemente de Android 10.

Aunque el sistema operativo Android como tal esté realizado en C/C++, las aplicaciones funcionan sobre una máquina virtual derivada de Java llamada Dalvik. Por ello, cuando se oye hablar de Android surge la relación directa con Java, ya que es el lenguaje de programación más usado para programar aplicaciones en este sistema operativo. Y aunque a día de hoy Google esté apostando fuertemente por la introducción de Kotlin como nuevo lenguaje de programación principal para Android, Java sigue siendo el más extendido y por tanto el que se usará para desarrollar esta aplicación.

¿Pero qué es Java? Java es un lenguaje de programación que sirve para desarrollar aplicaciones para gran cantidad de dispositivos. Java se basa en el concepto de POO (Programación orientada a objetos). En parte, esta es una de las ventajas de Java, la programación orientada a objetos se acerca bastante al pensamiento humano, permitiendo el desarrollo de grandes aplicaciones de una forma clara y organizada.

6.1.2. Android Studio

Android Studio es un entorno de desarrollo integrado, desarrollado por Google, que se impone como la herramienta oficial para el desarrollo de aplicaciones Android. Toma como lenguajes de programación principales Kotlin y Java, y dota al programador de una plataforma sencilla e intuitiva con numerosas ventajas dedicadas exclusivamente a la programación para hacer más positiva la experiencia.

Durante el desarrollo de los siguientes apartados se tratarán los diferentes conceptos y se intentarán aclarar las funciones que puedan resultar más complejas en mayor detalle. No se ha realizado una guía sobre este software ya que escapa del alcance del trabajo.

6.2. Introducción a las partes del proyecto

Cuando se afronta cualquier tipo de aplicación en un lenguaje de programación orientado a objetos, es importante tener en cuenta el concepto de modularidad. Se entiende por modularidad la descomposición de un sistema (aplicación) en un conjunto de módulos. Lo interesante de la modularidad es que los módulos estén poco acoplados, es decir, dependan poco del resto. Con esto se consigue un sistema de código fácilmente reutilizable y sostenible, que hace muy conveniente el uso de este tipo de organización.

Por tanto, a pesar de tratarse de una aplicación de gran tamaño en la que se haría imprescindible el uso del concepto anterior, por cuestión de orden y con la intención de abordar los problemas de forma aislada se perseguirá el uso de un código modularizado.

Por la forma en la que está organizado Android Studio se pueden observar dos partes diferenciadas

- Interfaz de usuario. La interfaz de usuario es el medio que permite al usuario comunicar información e interactuar con el sistema en cuestión, en nuestro caso la aplicación.
- Código de la aplicación. Se entiende como código de una aplicación al conjunto de instrucciones que abordan un objetivo común que es la funcionalidad final de dicha aplicación.

Aunque estas sean a primera vista las partes más evidentes, no se encuentran ni aisladas la una de la otra, ni son la unidad mínima a la que nos referimos cuando hablamos del concepto de modularidad. Es importante dejar claro que cuando se habla de módulo se habla de fragmento de código con una funcionalidad determinada, y en este caso ninguno de los anteriores cumple con dicho criterio.

Por tanto, la diferenciación que resulta más pertinente realizar es respecto a las distintas funcionalidades individuales que son necesarias para que la aplicación cumpla su función final. De esta forma se distinguen varios módulos a abordar que se presentan a continuación con una pequeña descripción que intenta explicar su relevancia.

- Interfaz de usuario. Se refiere tanto a la apariencia visual de la aplicación como a los distintos modos de interacción con el usuario. Este apartado es un entorno sencillo e intuitivo que intenta facilitar la experiencia del usuario.
- Gestor de fragmentos. Como se verá más adelante con el fin de favorecer la experiencia de usuario se ha basado la estructura en un "Tab Layout" con dos fragmentos. Esta vista y sus fragmentos tienen que ser gestionados y situados dentro de una actividad y de ahí la importancia de dicho gestor de fragmentos.
- Fragmento 1. Fragmento de recepción y gestión de datos recibidos desde el usuario. En este se encuentra la gestión directa de las entradas introducidas por el usuario y las funciones estrictamente relacionadas con el funcionamiento del mismo fragmento.

- Fragmento 2. Fragmento de muestra de datos recibidos. Se encarga de mostrar mediante una gráfica interactiva los datos recibidos al momento por medio del módulo de TCP.
- View Model. Estructura necesaria para hacer posible el paso de datos entre fragmentos. Actúa como una "nube" con distintas variables en las que ambos fragmentos pueden almacenar información, que al ser actualizada permite al resto de fragmentos reaccionar a dicho cambio.
- TCP Class. Módulo cuya función es proporcionar una plataforma para la recepción y el envío de datos por conexión TCP. Debe incluir funciones del tipo enviarMensaje, recibirMensaje, y otras funciones complementarias para conocer el estado de la conexión y para la interrupción de este.

Para dar una idea general de cómo va a ser la interfaz de usuario y mostrar la función que cobra cada fragmento sobre acciones directas sobre el motor se muestra la siguiente imagen.

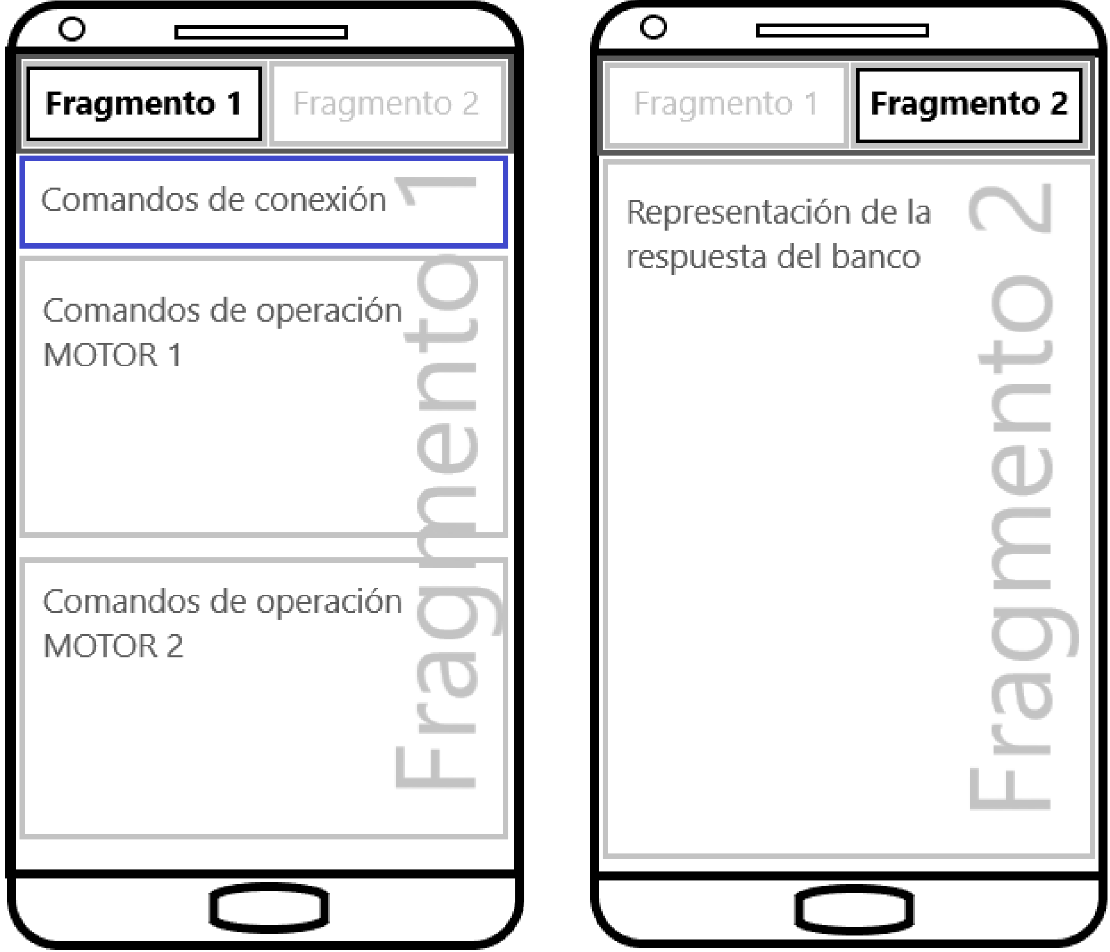


Ilustración 17. Imagen de la etapa de diseño de la aplicación. Fuente propia. En esta imagen se muestra una vista esquemática de ambos fragmentos, mostrando las funciones que se pretende implementar en cada área.

Como se puede observar en la imagen superior, el fragmento 1 (situado a la izquierda) se encarga de ejecutar los comandos de conexión, ejecutan las conexiones y desconexiones del servidor creado

por el PLC; y se el envío de comandos de operación sobre los motores (estos comandos se traducen en acciones en los motores como run, stop, cambios de velocidad…).

El fragmento 2 (situado a la derecha) se encarga únicamente de la representación gráfica de los datos recibidos de los motores. De esta forma se marca la diferencia entre la pantalla de manejo de motores y la de obtención y representación de datos. En los siguientes apartados se muestra la representación final de la interfaz de usuario (Ilustración 16).

A los módulos enumerados anteriormente se ha añadido un apartado de preparación del entorno. En este se incluyen ciertos recursos que se van a usar en los distintos módulos pero que no forman parte de ellos como tal. Pueden entenderse como ciertas herramientas que es necesario incluir para poder realizar el proyecto de forma satisfactoria.

En los próximos apartados se hace un análisis más detallado de los distintos módulos, viendo las funciones y partes más relevantes de cada módulo.

6.3. Preparación del entorno

Como se muestra en el apartado de introducción a Android Studio (5.1.2), ciertos apartados complementarios al código se encargan básicamente en la gestión de recursos. En los siguientes subapartados se muestran las modificaciones hechas y la utilidad de cada una de ellas.

6.3.1. Gradle

En este caso se modifica el fichero build.gradle (Module:App) al que se añaden un par de líneas de código a las dependencias de esta carpeta, que serán necesarias para implementar la gráfica interactiva en nuestra aplicación más adelante.

```
implementation 'com.jjoe64:graphview:4.2.2'
implementation fileTree(dir: 'libs', include: ['*.jar'])
```

La primera línea añade la librería de GraphView que, según la definición de sus creadores, se trata de “una librería para crear diagramas bonitos y flexibles por programa”.

La segunda línea de código añadida se trata de una dependencia binaria local, que se encarga de añadir la biblioteca añadida previamente al directorio ’libs‘ de la aplicación para que sea incluida correctamente.

6.3.2. Values

Como se ha mencionado, en la carpeta ‘values’ se encuentran ficheros XML que asignan a un ID un valor para luego ser llamado desde código, esta tarea optimiza tanto el código como la facilidad a la hora de modificar dichos valores. En este caso, sólo se le ha dado uso al documento strings.xml

que tiene almacenadas todas cadenas de texto que se usarán en la interfaz de usuario. Estas instancias siguen el siguiente formato:

```
<string name="nombre_id">Cadena de texto</string>
```

6.3.3. Drawable

Con la intención de hacer más sencillo e intuitivo el uso de una aplicación es buena idea el uso de imágenes que tienen asociado un significado. En este caso se han añadido imágenes correspondientes a start, stop, pause y alarm. Para ello, únicamente hay que localizar la ruta de la carpeta 'res/drawable' y arrastrar dichas imágenes dentro.

6.3.4. Manifest

Como se ha explicado previamente es necesario modificar el manifiesto de Android para solicitar los permisos para ciertas acciones. En este caso, solo es necesario añadir la petición de permiso para el acceso a la internet.

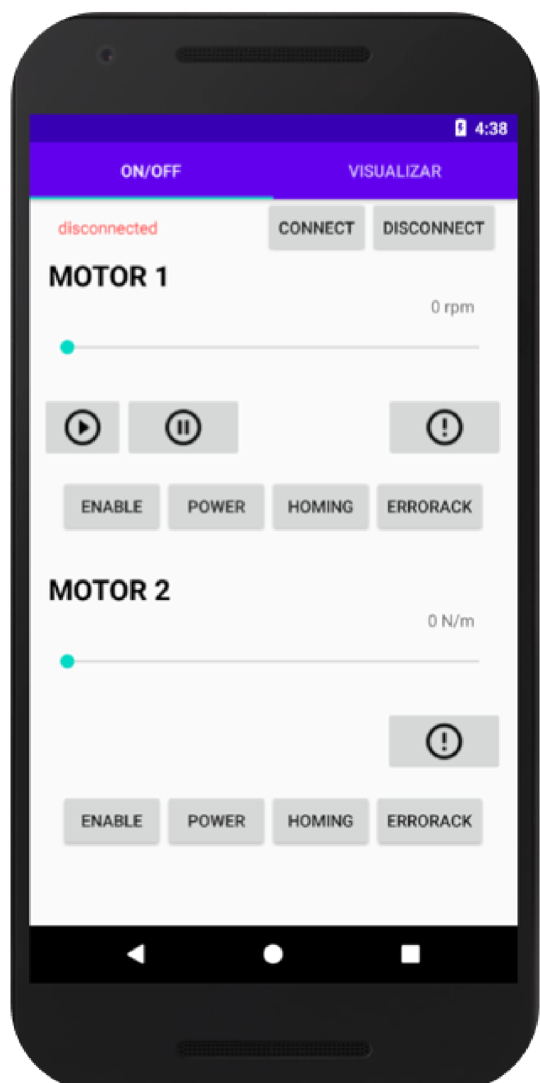
```
<uses-permission android:name="android.permission.INTERNET"/>
```

Con esa línea la aplicación ya es capaz de hacer uso de internet y por tanto ya existe la posibilidad de establecer conexión desde la aplicación con la red wifi generada por el PLC, que será usada para recibir y enviar datos.

6.4. Interfaz de usuario

La interfaz de usuario se trata de lo que Android llama TabLayout. Se trata de una estructura que te permite navegar entre vistas diferentes mediante el uso de pestañas o deslizamiento. Esta aplicación consta únicamente de 2 vistas entre las que navegar. Para facilitar el uso de este tipo de layout Android Studio da la opción de comenzar con una plantilla donde ya incluye las clases necesarias para su gestión, aunque esto se explica en el siguiente apartado, este está centrado únicamente en la parte visual de la aplicación.

A continuación, se muestran ambas vistas de la aplicación en funcionamiento.



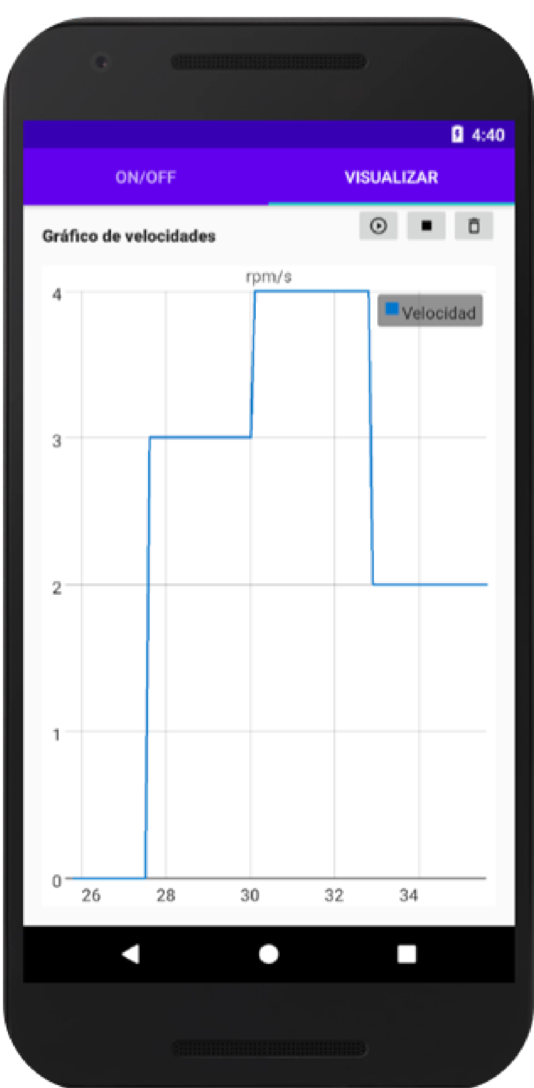


Ilustración 18. Vista de ambos fragmentos de la aplicación Android. Fuente propia. En esta se visualizan los dos fragmentos de la aplicación. En la vista izquierda se muestra el fragmento de acciones, mientras que a la derecha se muestra un ejemplo de representación gráfica del fragmento de visualización de datos.

Para lograr esa visualización modo TabLayout son necesarios tres ficheros de código, uno principal que viene por defecto al usar la plantilla proporcionada y dos secundarios de las dos vistas, que corresponden a dos fragmentos que han sido creados desde 0. En esta memoria se da una explicación de los distintos elementos que aparecen en pantalla, si se quiere conocer aspectos más precisos del código diríjase al anexo 1 en el que se encuentra todo el código de la aplicación.

Aunque la vista principal ya venga dada con la plantilla, conviene explicar su funcionamiento a nivel visual. La vista principal está diseñada con un tabulador en la parte superior que varía las separaciones dependiendo del número de fragmentos disponibles. Y el espacio bajo el tabulador está destinado a situar las vistas correspondiente con la pestaña seleccionada. Para ello se hace uso del viewGroup ViewPager que, además, hará posible pasar entre vistas deslizando lateralmente.

La vista 1, es la vista del fragmento de control del motor, en esta vista se quiere dar todo lo necesario para que el usuario sea capaz de realizar diversas acciones sobre ambos motores: una

completa inicialización de ambos, variar la velocidad y el par en los respectivos motores, frenado y arranque del motor 1 y paro instantáneo de alguno de los motores en caso de surgir una emergencia.

Para comprender como se ha creado esta vista hay que prestar atención a dos factores, los elementos usados y la situación de dichos elementos en pantalla.

Los elementos usados:

- Button. Se han incluido diez elementos de la clase Button. Esta clase crea botones a los que permite colocar un texto en su interior. Los elementos de esta clase tienen asociadas ciertas funciones como .OnClick(), lo que permite programar en código respuestas a las pulsaciones de dichos botones. El texto usado en estos botones será el colocado en las cadenas de texto introducidas en values/strings.xml mencionado anteriormente.

- ImageButton. Se ha hecho uso de cuatro elementos de la clase ImageButton. Se trata de una clase similar a la clase anterior pero esta permite incluir imágenes en lugar de texto. Los elementos también disponen del método .OnClick() para reaccionar a pulsaciones del mismo. Aquí se han incluido las imágenes introducidas previamente en res/drawable.

- TextView. Se ha hecho uso de cinco elemento de la clase TextView. Los elementos de esta clase permiten introducir texto dando distintas posibilidades para variar características como su estilo, tamaño, color o visibilidad. En este caso también se incluyen

- SeekBar. Se han incluido dos SeekBars. Los elementos de esta clase producen una barra de progreso interactiva, que permite su desplazamiento pulsando y arrastrando en pantalla. Esta clase tiene funciones que permiten reaccionar a cambios en la barra de progreso que permiten al código realizar funciones en función de la acción realizada.

Con respecto a la situación de dichos elementos en pantalla se ha hecho uso de un LinearLayout de orden vertical y varios LinearLayout horizontales contenidos. LinearLayout es una de las disposiciones que ofrece android para situar elmentos en pantalla. Su característica principal es que ordena de forma lineal los elementos que contiene ya sea colocándolos horizontal o verticalmente.

Por otro lado, la vista 2 es la vista que corresponde al fragmento de visualización, diseñado para visualizar con una gráfica, los datos de velocidad provinientes del motor a través de la comunicación TCP. Además, incluye tres botones para programar el paro y comienzo de muestreo y el borrado de datos y reset de la gráfica.

Respecto a los elementos usados:

- Button. Se han incluido tres elementos de esta clase con la finalidad de programar una función para el manejo de la gráfica.

- TextView. Además se añade un elemento de la clase TextView por fines estéticos que da título al gráfico.

- GraphView. Clase de la librería añadida previamente. Haciendo referencia al documento .xml de este fragmento adjuntado en el anexo 1, se observa que no se dan características visuales de la gráfica. Esto se debe a que debido a la naturaleza de la librería estas carácterísticas se introducen en el código de la aplicación.

Respecto a la forma de localización de los elementos en la pantalla, se ha usado una organización similar a la de la vista 1. Un LinearLayout vertical principal que contiene a dos LinearLayouts horizontales.

6.5. Gestor de fragmentos

Hasta el momento se ha hecho referencia al concepto de fragmento, pero no se ha explicado lo que es ni lo que implica. Y aunque no es parte del objetivo fundamental de la aplicación sí supone una característica relevante ya que condiciona el funcionamiento del código.

6.5.1. Definiciones

Primero es conveniente la definición de qué es un fragmento y una actividad.

- Un fragmento se refiere a una parte de la interfaz de una actividad que lo contiene. Esta actividad puede contener a uno o más fragmentos de forma simultánea. La explicación que se da en la guía de desarrolladores de Android es bastante aclaradora: “Puedes pensar en un fragmento como una sección modular de una actividad que tiene un ciclo de vida propio, que recibe sus propios eventos de entrada y que puedes agregar o quitar mientras la actividad se esté ejecutando (algo así como una “subactividad”…)”. Cabe mencionar que, aunque se puede hacer uso de numerosos fragmentos, en esta aplicación únicamente han sido usados dos.

- Una actividad (creado con clase “Activity”) es un componente dentro de una aplicación que permite al usuario interactuar con el dispositivo, esto no incluye una interfaz visual como la mencionada anteriormente, sino que incluye también la parte lógica y de manejo de datos. En una aplicación se pueden tener varias actividades e ir cambiando entre ellas, en la que una (la que aparece al inicio) se denomina como actividad principal. En este caso, sólo ha sido necesario el uso de una única actividad.

Una vez aclarados los conceptos, interesa ver la relación que existe entre fragmentos y actividad en el caso de esta aplicación, para entender el uso del gestor de fragmentos y del ViewModel que se explicará más adelante. Para esto se ha desarrollado el siguiente esquema explicativo.

6.5.2. Estructura relacional

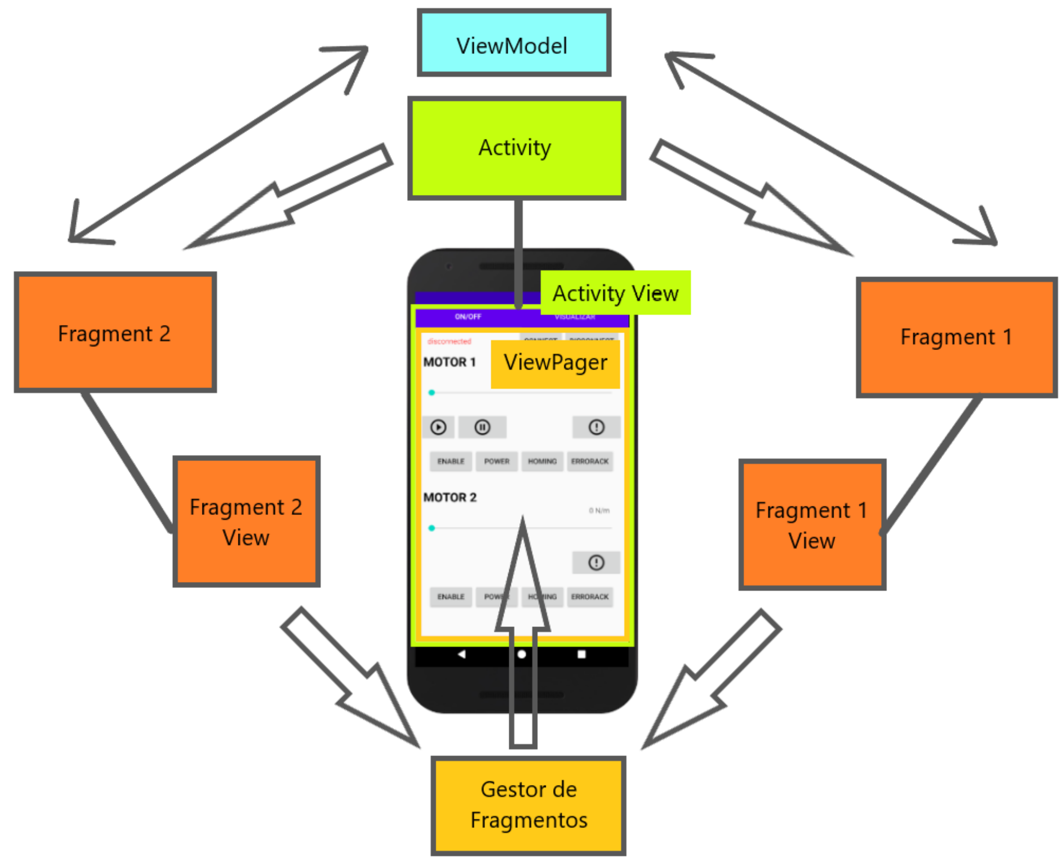


Ilustración 19. Relación de actividad principal y fragmentos de la aplicación Android. Fuente propia. En esta se muestran las relaciones entre los distintos módulos.

Este esquema muestra las relaciones entre los fragmentos, la actividad y sus vistas. Primero se da una explicación de las distintas relaciones presentes en el gráfico y posteriormente se explica el concepto de lo que se ha llamado "gestor de fragmentos" y las modificaciones que hay que realizar en el mismo para adaptarlo a esta aplicación en concreto.

Como se ha comentado anteriormente, se tiene una actividad única y principal que contiene dos fragmentos. Tanto la actividad como los fragmentos tienen una vista y un código lógico asociados. Que la actividad contenga a ambos fragmentos implica que estos tendrán que ser instanciados en algún momento desde la misma.

En lo que se refiere a la UI de la aplicación, se puede ver claramente que la actividad comporta toda la pantalla y que con el uso de un elemento de la clase ViewPager se da un espacio para añadir la vista de los fragmentos con el uso del gestor. Hay que destacar dos cosas, que el elemento ViewModel del gráfico se usa para la comunicación entre fragmentos y será desarrollado en el apartado 6.7. y que el gestor de fragmentos no se usa únicamente para situar la vista de los fragmentos en el ViewPager, sino que también es necesario para instanciar los fragmentos dentro de la actividad.

6.5.3. Gestor de fragmentos

Cuando se ha hablado de gestor de fragmentos se hacía referencia a las 3 clases que otorga Android Studio para gestionar los fragmentos tanto a nivel visual como a nivel interno.

Estas son PageViewModel, que actúa como extensión de la clase ViewModel, PlaceholderFragment que se trata de una extensión de la clase Fragment, y SectionsPagerAdapter que se trata de una extensión de FragmentPagerAdapter. Si se quiere profundizar más en estas clases, sus extensiones y su funcionalidad, se recomienda echar un vistazo al código de la aplicación en anexo 1 y en el enlace adjunto de la guía de desarrolladores que hace una revisión más en detalle de los mismos. Ya que en este apartado se mostrarán únicamente los cambios realizados para ajustar estas clases a la aplicación y algunas instancias para incluir a los fragmentos.

En el PlaceholderFragment se debe cambiar la función newInstance (int index), donde se deben sustituir los casos del switch por los fragmentos creados que se van a crear e igualarlos a la variable fragment que se retornará.

```
public static Fragment newInstance(int index) {
    Fragment fragment = null;

    switch (index) {
        case 1: fragment = new InitFragment(); break;
        case 2: fragment = new VisualDataFragment(); break;
    }

    return fragment;
}
```

En este caso, se ha añadido el nombre de los fragmentos creados, InitFragment() es como se ha llamado al fragmento de control y VisualDataFragment() es el nombre que se le ha dado al fragmento de visualización.

Por otro lado, en la clase SectionsPageAdapter hay que realizar dos modificaciones; se deben incluir los nombres que quiere que aparezcan en las pestañas superiores en la variable TAB_TITLES.

```
private static final int[] TAB_TITLES = new int[]{R.string.tab_text_1,
R.string.tab_text_3};
```

Además, se debe especificar en la función getCount() que el número de páginas a gestionar son dos.

```
public int getCount() {
    // Show 2 total pages.
    return 2;
}
```

Con estos dos cambios ya está ajustado este gestor de fragmentos al caso en particular. Cabe apuntar que la relación entre el MainActivity y los fragmentos se da a través del ViewPager. Instanciándose el ViewPager desde esta actividad principal “MainActivity” y conteniéndose en éste ambos fragmentos.

6.6. Fragmento 1 – InitFragment(…)

Este apartado está dedicado únicamente al código y no a la parte visual que ya ha sido explicada en el apartado 5.4.' Interfaz de usuario'. Este primer fragmento se ha llamado InitFragment(….) ya que se trata del fragmento que aparecerá en pantalla al iniciar la aplicación, corresponde a lo que se hacía referencia en el apartado 6.3. como fragmento de control.

Este fragmento tiene la función principal de interpretar las instrucciones introducidas por pantalla y reaccionar a ellas de forma pertinente, ya sea realizando una acción, actualizando la interfaz de usuario o ambas. Se va a dividir el código en diferentes subapartados para explicar las principales funcionalidades desarrolladas.

6.6.1. onCreateView(…)

Android provee de unos métodos para realizar acciones en las distintas etapas por las que pasa una aplicación, onCreateView(…) es uno de ellos. El código dentro de este onCreateView(…) se ejecutará inmediatamente después de la creación de la vista de la actividad principal que lo contiene. Siendo este el primer momento que en el que se muestra la vista, resulta conveniente realizar acciones relacionadas con el reconocimiento de lo que se está mostrando en pantalla y que pretende ser usado o modificado en el código.

Por tanto, se procede a crear variables para cada uno de los elementos de las clases Button, ImageButton, Seekbar y los TextView que van a ser modificados, los que se encuentran sobre las seekBars mostrando el progreso y el texto en la parte superior que indica lo conexión. Una vez creadas estas variables hay que relacionarlas con la vista en pantalla con el uso de findViewById:

```
btnHoming1 = vista.findViewById(R.id.btn_homing_1);
```

En el caso mostrado en particular, se relaciona el botón de homing del motor 1 con que tiene un id "btn_homing_1" con la variable creada en el código para dicho botón "btnHoming1". Esto se hace para todos los casos mencionados en el párrafo anterior. De esta forma ya se pueden usar dichas variables para hacer modificaciones o acciones a partir de las acciones del usuario u otros eventos.

6.6.2. seekBar.setOnSeekBarChangeListener

El elemento de la clase seekBar por sí mismo no realiza ninguna acción, pero esta clase trae aparejada una función que permite reaccionar a cambios introducidos por pantalla. Esta función se llama setOnSeekBarChangeListener, e incluye tres métodos que permiten realizar acciones justo cuando se está desplazando, cuando antes de empezar a mover el punto de progreso y justo al soltar la barra. De los tres disponibles solo se han usado los dos siguientes:

```
onProgressChanged(SeekBar seekBar, int progress, boolean fromUser){…
```

El método onProgressChanged(…) se activa conforme se desplaza cualquiera de las seekBars, debido a esto resulta interesante para realizar un par de acciones:

- Convertir y almacenar en una variable extendida al espectro del fragmento el valor equivalente del progreso de la barra. Cuando se activa este proceso, el progreso de la barra en cada momento se pasa al programa que va de 0/100, este rango se traduce al rango de velocidad o par del motor y se almacena en una variable para enviarlo posteriormente.

- Actualizar la interfaz de usuario, como únicamente con la seekBar es complicado saber el valor exacto que se está enviando, se ha hecho uso de un textView que se cambia al instante conforme se varía dicho progreso, con el uso de la variable que se ha almacenado.

```
onStopTrackingTouch(SeekBar seekBar) {…
```

Por otro lado, se ha hecho uso del método onStopTrackingTouch(…). Este, como su nombre indica, se activa una vez el usuario para de desplazar la SeekBar. Este evento se ha aprovechado para "traducir" el valor que se ha almacenado en la variable durante el movimiento de la barra a un código que será posteriormente interpretado por el PLC y, enviar dicho código por TCP al mismo.

6.6.3. OnClick(…)

Una vez las variables se encuentran relacionadas con los elementos Button en pantalla, falta programar la respuesta en caso de que estos se pulsen. Esto se puede hacer de forma individual como se ha hecho con las seekBars, pero en este caso al ser múltiples botones interesa programar los botones para que reaccionen ante pulsación instanciando un método común onClick y distinguir dentro de este método cuál de ellos ha sido pulsado.

Para que los botones produzcan dicha reacción al ser pulsados se coloca un .setOnClickListener de la siguiente forma:

```
btnConnect.setOnClickListener(this);
```

De esta forma, al pulsar el botón Connect ya se ejecutaría el código dentro de método onClick(…). Se ha hecho lo mismo para todos los botones y se ha gestionado posteriormente dentro del onClick(…) de la siguiente forma.

```
public void onClick(View v) {
    switch (v.getId()) {
        case R.id.btn_connect:
            // Código relacionado con dicho conexión

            break;
        case R.id.btn_disconnect:
            // Código para realizar conexión
```

```
            break;
        case …
```

Como se puede observar, el switch coge el id del botón pulsado en la vista v (la del fragmento) y ejecuta el código el caso con el que coincida dicho id.

Respecto a los códigos que se llevan a cabo dentro de cada botón cabe destacar dos casos especiales de dos botones en particular y el caso general que se usa para el resto de botones:

- Button Connect. Al pulsar el botón Connect se crea un hilo (thread) sobre la variable de la clase StartClient() que se explica en el apartado 5.9. TCP Class. Se empieza dicha amenaza habilitando la recepción y el envío de datos en segundo plano y se activa la función stateConnection que actualizará si la aplicación se encuentra conectada o no.

- Button Disconnect. Al pulsar el botón Disconnect se ejecuta la función stop() de la clase StartClient cerrando el socket y por tanto eliminando la recepción o el envío de datos.

- Resto de botones. El resto de botones constan de un código bastante similar, se basan en enviar un valor al PLC el cual ejecuta la acción asociada a dicho valor. Para ello, se usa la función sendMessage() de la clase StartClient que también se muestra más adelante.

6.6.4. stateConnection

Esta función de tipo Runnable consulta continuamente una vez iniciado el estado de la conexión. De forma sencilla el tipo Runnable permite el desarrollo de un código en segundo plano al mismo tiempo que se ejecuta la aplicación de forma normal. En este caso, se ha hecho uso de un Handler que sirve para operar dicho Runnable, permitiendo frenarlo, ejecutarlo cada cierto tiempo y otras opciones.

Esta función es bastante simple, consta de un if que confirma si se cumple que la conexión sigue abierta y cambia el texto y el color del texto en función de si sigue abierta.

```
boolean isClosed = startClient.isClosed();
    if (isClosed||!thread.isAlive()) {
        // Se cambia el texto a Disconnected en rojo
    } else {
        // Se cambia el texto Connected en verde
    }
connectHandler.postDelayed(stateConnection, 10);
```

Se puede observar que se hace uso de la función .isClosed() que muestra si el socket ha sido cerrado intencionadamente. Además, al terminar el código recurre al handler creado para instanciarlo de nuevo 10 ms más tarde.

Dentro de este fragmento existen un par de funciones más que no se han mencionado, ya sea por no tener una relevancia directa sobre el funcionamiento de la aplicación, o por estar estrechamente relacionados con uno de los apartados siguientes y será mencionado más adelante.

6.7. Fragmento 2 – VisualDataFragment()

De nuevo, cabe mencionar que en este apartado se hace referencia al código lógico del fragmento y no a su vista asociada que ya ha sido desarrollada en apartados anteriores. Este fragmento ha recibido el nombre de VisualDataFragment(…) que hace honor a su función de mostrar los datos recibidos por conexión de forma gráfica. A continuación, se introducen las funciones más relevantes desarrolladas dentro de este fragmento.

6.7.1. OnCreateView(…)

De nuevo se hace uso de este método para crear las variables y relacionarlas con los botones y el gráfico de pantalla con la intención de reaccionar a ellos. Además, se configura la graphView añadida en el layout del fragmento, cambiando ciertos valores que modifican la apariencia y se habilitan ciertas características del uso.

Una parte importante a comentar es la siguiente:

```
series = new LineGraphSeries<>(new DataPoint[]{
});
velChart.addSeries(series);
```

En estas dos líneas, se crea una serie lineal a la que se le añadirán elementos de la clase DataPoint. La serie lineal creada se configura como serie a representar gráficamente al utilizar la función addSeries(…) sobre la variable que se había relacionado con la gráfica. Posteriormente, conforme se añadan puntos a dicha serie se irán actualizando en el gráfico.

6.7.2. PlotData

PlotData es una función de clase Runnable lo que, como ya se ha mencionado antes, tiene la capacidad de funcionar al mismo tiempo que se realizan otras actividades. Aquí se hace uso de nuevo de un Handler llamado plotDataHandler para manejar la función plotData.

Básicamente, la función realiza lo que su nombre indica, añade puntos a la serie de la gráfica con un tiempo de muestra predefinido. La parte más relevante de la función es la siguiente:

```
plotDataHandler.postDelayed(plotData, SAMPLE_TIME);

series.appendData(new DataPoint(time, nextPoint), true, 100);
…
…
time += (float)SAMPLE_TIME/1000;
```

La primera línea tiene la función de reactivar la función en el tiempo SAMPLE_TIME que es una constante que se fija al comienzo del documento. Al invocarse nada más comenzar la función se asegura de que dicho SAMPLE_TIME se cumplirá de forma precisa.

La segunda línea añade un nuevo punto a la serie que está relacionada con el gráfico. Este punto tiene como entrada en x el tiempo que se actualiza al final de la función dependiente de la constante SAMPLE_TIME, y como entrada en el eje y la variable entera nextPoint que se actualiza al recibir un nuevo dato. Las dos últimas entradas habilitan que la función se vaya desplazando al añadir datos y fijan que haya un número de 100 puntos como máximo en pantalla.

Esta función está rodeada por un 'if' que sólo se lleva acabo si la variable Booleana runActivated es verdadera. La cual será manipulada en la pulsación de los botones.

Dentro de la función hay unas líneas de código dedicadas a perfeccionar la fluidez de la representación, pero no tiene importancia en el funcionamiento como tal, si se tiene interés en comprobarlas estarán añadidas en el Anexo 1.

6.7.3. OnClick(…)

Se ha decidido comentar este apartado tras explicar la función plotData para comprender mejor de qué forma influyen los cambios en estos en el funcionamiento normal de la gráfica. Se ha seguido la misma estructura de switch seguida en el fragmento 1. Dándose tres casos distintos que resultan interesantes de comentar.

```
case R.id.btn_start_chart:
    runActivated = true;
    plotDataHandler.postDelayed(plotData, SAMPLE_TIME);
    break;
```

Al pulsar el botón con el símbolo de start, se pone positiva la variable runActivated y se llama a la función con el uso del Handler mencionado anteriormente. De esta forma la función plotData() entra en el ciclo de muestro explicado en el apartado anterior.

```
case R.id.btn_pause_chart:
    runActivated = false;
    break;
```

Al pulsar el botón con el símbolo pause, se niega la variable runActivated haciendo que pare el muestreo. Cabe mencionar que por la configuración que se ha dado a la gráfica tras pulsar pause es posible hacer acciones sobre la gráfica como hacer zoom o desplazarse a datos anteriores.

```
case R.id.btn_delete_chart:
    runActivated = false;
    series.resetData(new DataPoint[]{});
    time = 0;
    velChart.getViewport().setMinX(0);
    velChart.getViewport().setMaxX(TIME_RANGE);
    break;
```

Por último, al pulsar el botón con el símbolo de la papelera, se niega el runActivated parando el muestreo y se eliminan los datos tomados retornando el valor de la gráfica a su estado inicial.

6.8. TCPProcess - StartClient

Hasta el momento se han hecho referencias continuas a la recepción y envío de datos por conexión TCP. ¿Pero cómo se gestiona este envío y recepción de datos? Para resolver esta gestión se ha realizado una nueva clase, que se implementa a partir de la clase Runnable, que incluye las funciones necesarias para ello. Aunque a nivel visual no tenga una repercusión directa es un punto relevante del proyecto ya que es la forma con la que comunicarse con el PLC que controla los motores.

Aunque se han creado un par de funciones adicionales que ayudan a gestionar la desconexión y la consulta de estado, se trata de funciones sencillas sin mayor relevancia por lo que no serán exploradas en profundidad en este documento. A continuación, se incluyen dos apartados que pretenden describir las dos funcionalidades principales de esta y las funciones que intervienen en ellas.

6.8.1. Conexión y recepción de datos

En este proceso intervienen dos funciones principalmente, una que se encarga de crear el socket de conexión y leer los datos que se van recibiendo por dicho socket, y una segunda instanciada desde la anterior que contribuye a que los datos puedan llegar a la variable nextPoint en el fragmento 2 y representarse gráficamente. A continuación, se describen ambas en detalle.

La primera está dentro del mecanismo run() del Runnable implementado en la clase, esto implica que empezará a funcionar cuando se cree y se inicie un hilo (Thread) sobre una variable de la clase en cuestión (StartClient). Ya se ha mencionado que esto ocurre en el fragmento 1 al pulsar connect. Dentro de este run lo principal es lo siguiente:

```
socket = new Socket(IP, PORT);

while (!Thread.currentThread().isInterrupted()) {
    DataInputStream inputStream = new DataInputStream(socket.getInputStream());
    int message = inputStream.readInt();
    InitFragment.getInstance().showMessage(message);
}
```

Al iniciar la amenaza se crea un nuevo socket con las constantes IP y PORT, que deben coincidir con las fijadas en el PLC. Y mientras que la amenaza creada no sea interrumpida, se creará una entrada de datos que corresponde con la entrada de datos del socket creado, se leerá el valor en formato int entrante por esta línea y se instanciará la segunda función (que está presente en el fragmento 1) con el mensaje de entrada recibido. Hay que tener en cuenta que esta lectura se realizará de forma continua, en segundo plano, mientras no se interrumpa la amenaza.

Se muestra a continuación lo que realiza está segunda función que tiene un cometido muy simple. Esta segunda función es la función showMessage que tiene como entrada una variable entera message, esta también ejecuta un Runnable, la idea de este Runnable es aprovechar que funciona en segundo plano para no hacer más lento el ciclo de recepción de datos. Es decir, al ser instanciada el programa anterior sigue su curso mientras esta realiza las funciones pertinentes. El código central de esta función es el siguiente:

```
if (receivedMessage != message){
receivedMessage = message;
viewModel.setNewValue(message);}
```

Simplemente, el código filtra si se ha recibido un mensaje distinto al anterior y en ese caso, hace un set de dicho mensaje en la variable viewModel de la clase SharedViewModel. La función de esta clase SharedViewModel se encuentra descrita en el apartado 5.9. SharedViewModel, pero se adelanta que cumple la función de llevar este dato al fragmento 2 para que sea posible su representación.

6.8.2. Envío de datos

Para el envío de datos sólo se ha necesitado realizar una función definida dentro de la clase StartClient, al contrario que la función de recepción, esta únicamente envía datos cuando se instancia. Esta función es sendMessage(…) que toma como entrada una variable entera message.De nuevo se trata de una función de tipo Runnable con la intención de que funcione en segundo plano, esto se debe a que este tipo de operaciones suelen necesitar más tiempo y pueden afectar a la experiencia del usuario, dando en algunos casos fallos en la aplicación.

La aplicación comprueba que existe un socket creado y en caso de existir ejecuta dos líneas de código que realizan el envío del mensaje.

```
DataOutputStream outputStream = new DataOutputStream(socket.getOutputStream());
outputStream.write(message);
```

Como se observa el funcionamiento es similar al de la recepción, se crea una línea de salida que se corresponde con la línea de salida del socket definido y sobre esta línea se escribe el mensaje que se pretende enviar.

6.9. SharedViewModel

Finalmente, queda abordar una última clase que, aunque se ha mencionado puntualmente, no se ha explicado cuál es la necesidad de la creación de esta. El problema surge del uso de fragmentos, los fragmentos (Como se observa en la Ilustración 16) pueden comunicarse con la actividad que los contiene, pero no entre ellos directamente. En el caso de esta aplicación, la comunicación TCP se ha definido en el fragmento 1, pero los datos recibidos por dicho fragmento necesitan ser usados en el fragmento 2.

Hasta hace poco, la solución para la comunicación entre fragmentos de la misma actividad era crear una interfaz en la actividad que las contenía que se encargase de esto. Pero recientemente, Android Studio incluyó una nueva clase que permite el intercambio de datos entre fragmentos de forma reactiva y relativamente sencilla. Con esta nueva clase el flujo de los datos recibidos queda así:

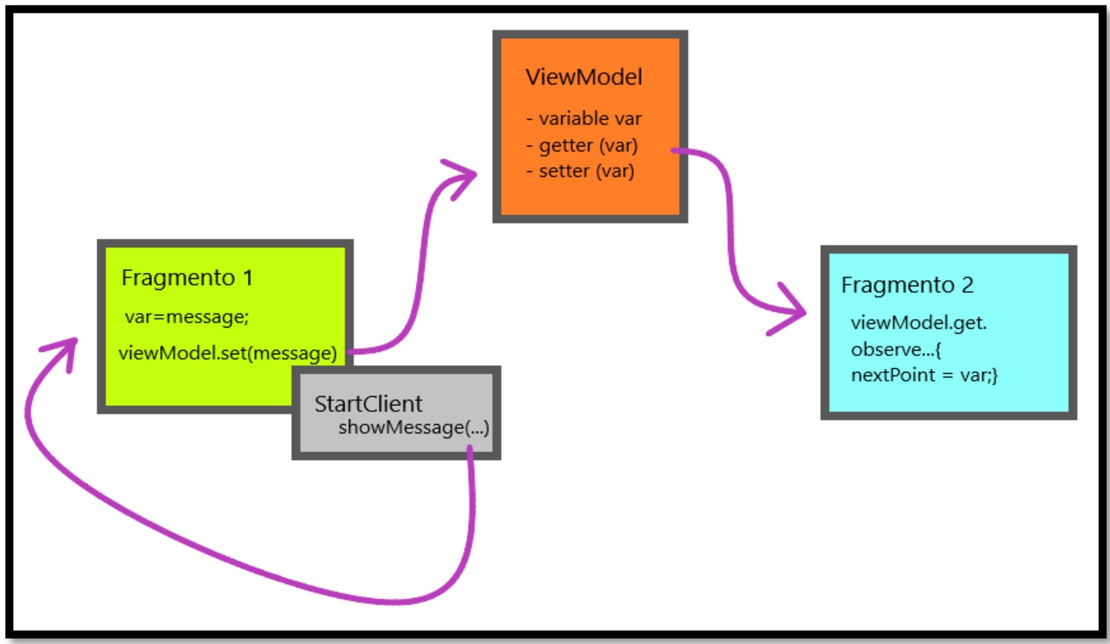


Ilustración 20. Muestra del flujo de datos de estado del motor entre fragmentos. Fuente propia. El flujo entra por el fragmento 1 y para pasar al fragmento 2 es necesario el uso de una interfaz ViewModel que permita el intercambio de datos entre ambas.

La imagen pretende dar una idea sencilla de forma gráfica del flujo de un dato recibido hasta que llega a la variable nextPoint en el fragmento 2. Para crear este flujo es imprescindible la creación de una clase extensión de viewModel e introducir en este punto la variable con las respectivas funciones para su gestión. La clase se ha llamado SharedViewModel y en conjunto ha quedado así:

```
public class SharedViewModel extends ViewModel {
    private MutableLiveData<Integer> receivedData = new MutableLiveData<>();

    public void setNewValue(Integer newValue){
        receivedData.setValue(newValue); }

    public LiveData <Integer> getNewValue() {return receivedData;}
}
```

En la primera línea de código vemos la creación de una nueva clase extendida de la clase ViewModel. A esta se le añade una variable MutableLiveData en este caso entero (Integer), esto se ha hecho de esta manera por costumbre, aunque en este caso podría haberse declarado directamente como un entero. Y para esta variable se añaden dos funciones:

- setNewValue. La función actúa como un 'setter' típico, permite la introducir el valor de la variable receivedData.
- getNewValue. Aunque podría parecer una función 'getter', esta función tiene la peculiaridad que el al estar instanciada como "public LiveData…" se puede observar el valor del 'return' de la función y reaccionar a su cambio, característica que resulta útil para que al recibir un nuevo dato por TCP y cambiar el valor de la variable, observar este cambio e incluirlo en la variable nextPoint del fragmento 2.

Una vez se ha creado la clase a partir de ViewModel, sólo queda instanciarla en ambos fragmentos para que se relacionen al mismo ViewModel y preparar el método '.observe' para reaccionar a cambios en la variable.

Como se ha mencionado se ha creado una variable tipo SharedViewModel y se ha definido de la siguiente forma en cada uno de los fragmentos.

```
SharedViewModel viewModel= new ViewModelProvider(requireActivity())
                                          .get(SharedViewModel.class);
```

Además, para hacer que el fragmento 2, actualizase la variable nextPoint se ha programado la funcionalidad '.observe' mencionada anteriormente, quedando así.

```
viewModel.getNewValue().observe(getViewLifecycleOwner(), new
Observer<Integer>() {
    @Override
    public void onChanged(Integer integer) {nextPoint = integer;}});
```

Se observa que se programa con la función '.observe' y se añade un método onChanged que pasa el valor, lo único que queda es almacenarlo en la variable que tiene de 'scope' todo el fragmento 2 para poder usarlo para cambiar la gráfica.

7. CODIGOS DE COMUNICACIÓN

7.1. Introducción

Se hace referencia a códigos de comunicación a los valores que se van a enviar y recibir, representando las órdenes y la información que realmente se quiere comunicar. Por qué son necesarios estos códigos.

Primero, es necesario “traducir” las órdenes a binario para poder comunicarlas, es decir, no es eficiente mandar una cadena que diga “encender” para ejecutar la orden encender. Por tanto, dichas órdenes que se deben ejecutar al pulsar un botón se asociaran con un número binario para enviarlas por TCP.

Además, se están utilizando dos órdenes que involucran valores enteros, cabe pensar que estos podrían enviar su valor directamente al ser enteros. Si se plantea, por ejemplo, que se recibe un 2000 por TCP ¿Qué indica? ¿Qué se ponga a 2000 rpm el motor 1? ¿Qué se aplique un par resistente de 2000 N/m? Así se ve claro el problema, hay que añadir algún tipo de identificador en el código que distinga entre si el dato recibido pretende cambiar el par o la velocidad, o de si se trata de una orden de botón.

Segundo, hay que considerar que la velocidad de giro debe enviar continuamente de los motores a la aplicación. En este caso, sí se podría enviar directamente con su valor en rpm ya que la aplicación planteada no incluye una medida al momento del par. Sin embargo, se dejará un “espacio” por si en un futuro se quiere realizar una ampliación y comenzar a pasar valores de par reales a la aplicación.

7.2. Códigos de orden

Se hará referencia a códigos de envío como todo aquel que recibe el PLC y que se traduce en un cambio en el estado del motor. Esto incluye las acciones que contribuyen a la puesta en marcha del motor y su arranque y paro (Power, Homing, ErrorAck, Enable, Play, Stop y Emergencia), además de las que permiten la variación de los parámetros de velocidad y par.

Por como está planteada la plantilla TCP del Automation Studio, la recepción de datos se da en paquetes de 16 bits. Esto limita el espacio de trabajo para dichos códigos de envío, pero es suficiente para esta aplicación.

Para distinguir entre los distintos tipos de datos que se reciben en Automation studio se va a hacer uso del bit de mayor peso. En caso de ser ‘1’ el bit de mayor peso el dato correspondiente será un dato de la nueva velocidad o par. Por tanto, en caso de ser ‘0’, será una orden relacionada con un pulsado de botón.

En caso de ser el Bit de mayor peso ‘1’, es decir que se pretenda cambiar el par o la velocidad, el segundo bit de mayor peso será usado para discernir entre si el dato recibido se trata de una nueva velocidad o un nuevo par. Por tanto, el esquema de bits se puede esquematizar de la siguiente forma:

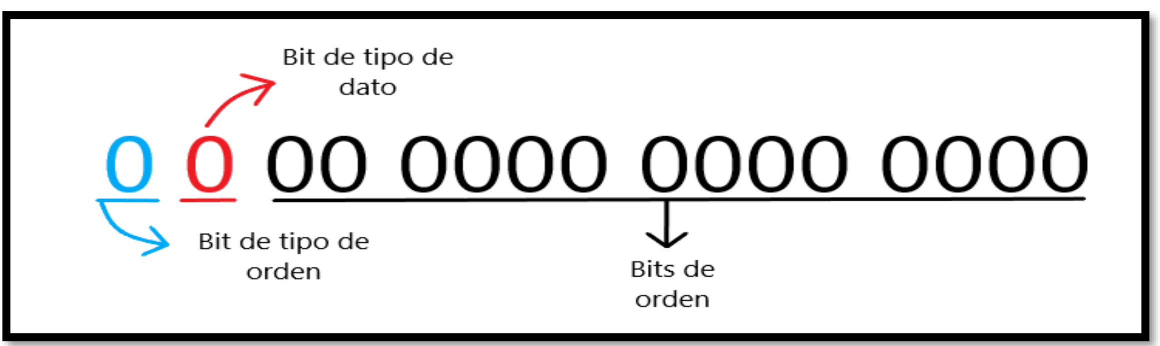


Ilustración 21. Uso de los bits de comunicación. Fuente propia. Se detallan los distintos bits de la palabra pasada y su uso.

Ahora queda determinar el uso de los bits de orden. Estos 14 bits restantes valdrán para definir las distintas posibles órdenes. En la siguiente tabla se muestra el valor que debe tomar cada uno de los bits para las distintas órdenes de botón.

Orden	**Código (14 bits de orden)**
Enable 1	00 0000 0000 0001
Enable 2	00 0000 0000 0010
Emergency 1	00 0000 0000 0011
Emergency 2	00 0000 0000 0100
Power 1	00 0000 0000 0101
Power 2	00 0000 0000 0110
Homing 1	00 0000 0000 0111
Homing 2	00 0000 0000 1000
ErrorAck 1	00 0000 0000 1001
ErrorAck 2	00 0000 0000 1010
Stop	00 0000 0000 1011
Run	00 0000 0000 1100

Tabla 1. Códigos de comunicación.

En el cuadro se muestran los códigos de las ordenes de botón. Los dos bits que preceden a dicha cadena de bits son 00. Como se observa al tener configurados un número de órdenes tan reducido hay muchos bits que se encuentran inutilizados.

Ahora toca debatir cómo se va a realizar el envío de los enteros de velocidad y par. Con 14 bits se puede enviar valores entre 0 y 16384. Aunque este valor cubra la velocidad en rpm máxima que se puede aplicar al motor, no cubre el par máximo aplicable. Sin embargo, esto no supone un problema, al introducir los valores con el uso de una seekBar (siendo el objetivo una interfaz de usuario más amigable) el progreso se da en 100 escalones de 0 al valor máximo. Es decir, si la velocidad máxima fuese de 15000 rpm, los datos irían de 0 a 15000 de 150 en 150. Cabe aclarar que se hizo de esta manera ya que en este proyecto no había ningún interés en conseguir la precisión de un incremento unitario, pero sí que era relevante la interactividad que el usuario pudiese tener con la app.

Por tanto, no es necesario mandar el valor completo de rpm, basta con sumar el valor del progreso que tiene de rango [0, 100] y convertirlo al llegar a Automation studio. Para ello solo se necesitan los últimos 7 bits para mandar el máximo valor de par o velocidad.

7.3. Códigos de datos

En el caso de la comunicación del estado del motor, sí que se busca precisión en la medida de 1 rpm, para poder apreciar adecuadamente la respuesta. Por ello ya no se puede pasar el dato de forma porcentual y se debe pasar en rpm por lo que va a hacer uso activo de más bits.

En este caso se vuelve a pasar una variable de 16 bits. El primer bit se reserva por si se hace una ampliación del proyecto en un futuro, de esta forma dicho bit sería el bit para alternar entre par y velocidad. Con 15 bits se puede representar hasta 32767 (2^{15}-1), lo que abarca la velocidad máxima posible del motor.

8. MANUAL DE ARRANQUE

8.1. Introducción

Aunque la aplicación es bastante sencilla y su uso resulta bastante intuitivo, se ha realizado un manual de usuario que explica cómo dar los pasos básicos para el uso de esta aplicación.

Se va a comenzar indicando las señales visuales que se pueden sacar de la observación del banco de ensayos, qué significa cada una de ellas y cómo se debe actuar en cada una de ellas. A continuación, se va a guiar un proceso desde cero a poner los motores en funcionamiento y su posterior apagado. Por último, se va a mostrar cómo se debe hacer uso de la función de graficado de velocidad y las funciones que ofrece.

8.2. Interpretación de señales

Aunque la aplicación Android no muestra el estado en el que se encuentra el sistema de motores, el sistema de luces del ACOPOS P3 da una información muy valiosa sobre el estado en el que se encuentra el motor en cada momento.

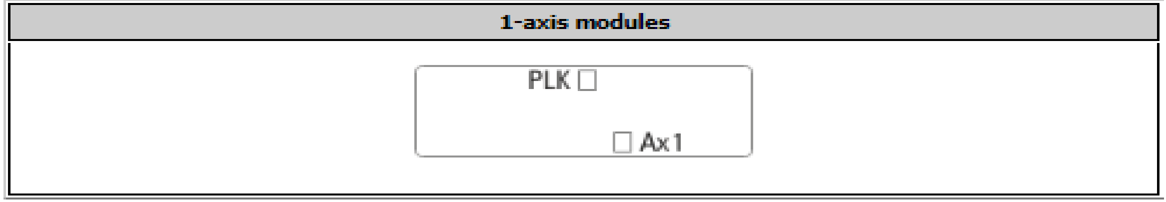


Ilustración 22. Displays de estado del servoaccionamiento. Imagen de la guía de ayuda sección Hardware/ Motion Control/ ACOPOS P3/ 8EI servodrives / Status indicators.

El ACOPOS P3 tienes dos displays led (ya que se trata de módulos de un único eje). El primero es el módulo del PLK; este variará principalmente al encender el dispositivo y mientras el programa se inicia. La mayoría de errores señalados por ese display tienen que ver con errores de código o de conexión, por lo que una vez el programa está optimizado y las conexiones configuradas no da una información relevante para el usuario.

El segundo display marcado por Ax1, representa el estado del eje y este sí que da una información valiosa del estado del eje. La siguiente es una tabla adaptada de las posibles señales de dicho link y su posible significado.

Color	Función	Descripción	
Verde	Preparado	Verde fijo	Módulo preparado para comenzar su funcionamiento.
		Verde Parpadeo	Módulo no preparado para comenzar su funcionamiento. Este estado se deberá principalmente a que no se ha dado la señal enable del motor. También representa ciertos errores de conexión y operación. Ver guía.
Rojo	Error	Rojo fijo	Error permanente en el módulo. Ej. Sobrecorrientes permanentes.
		Rojo parpadeo	El módulo se está quemando.
Naranja	Encendido	Naranja Fijo	Power activado satisfactoriamente.
---		Led apagado	No hay suministro de voltaje al módulo en cuestión.

Tabla 2. Señales de funcionamiento de los servoaccionamientos ACOPOS P3.

Durante el funcionamiento normal de la aplicación solo se pasará por Apagado/ Verde parpadeo/ Verde fijo/ Naranja fijo. Los demás estados de este display se deben a errores permanentes, para más información sobre los mismos diríjase a la guía.

Conociendo los estados que los leds indicadores pueden tomar, ya se tiene una idea de lo que está ocurriendo en cada momento. En el siguiente apartado se hace uso de esta información para arrancar y apagar correctamente el banco de motores.

8.3. Conexión App – Banco de ensayos

Este es el primer paso desde el punto de vista del usuario. Para ello el banco de ensayos tiene que estar conectado a la red, de esta forma el programa cíclico del PLC comienza a funcionar y se establece el servidor TCP, además el router establece la red por la que la app se debe conectar.

Desde un dispositivo con la aplicación descargada hay que entrar en la red que ha establecido el router, esta está configurada sin contraseña por lo que no hay que dar ningún tipo de identificación.

Una vez establecida la conexión wifi hay que entrar en la aplicación y usar el botón connect. La aplicación contiene los datos de IP y puerto donde está establecido el servidor TCP. Si el socket se ha creado correctamente y la conexión se encuentra establecida, aparecerá la señal en verde con el mensaje “conected”.

Al terminar con el uso de dicha aplicación se puede hacer uso del botón disconnect que interrumpe el flujo de datos cerrando el socket. De esta forma se evita un envío de datos indeseado.

8.4. Puesta en marcha – Apagado

En este apartado se explica los pasos a seguir para una correcta puesta en marcha y apagado de los motores. La siguiente imagen muestra los pasos a seguir para un correcto encendido y apagado de los motores.

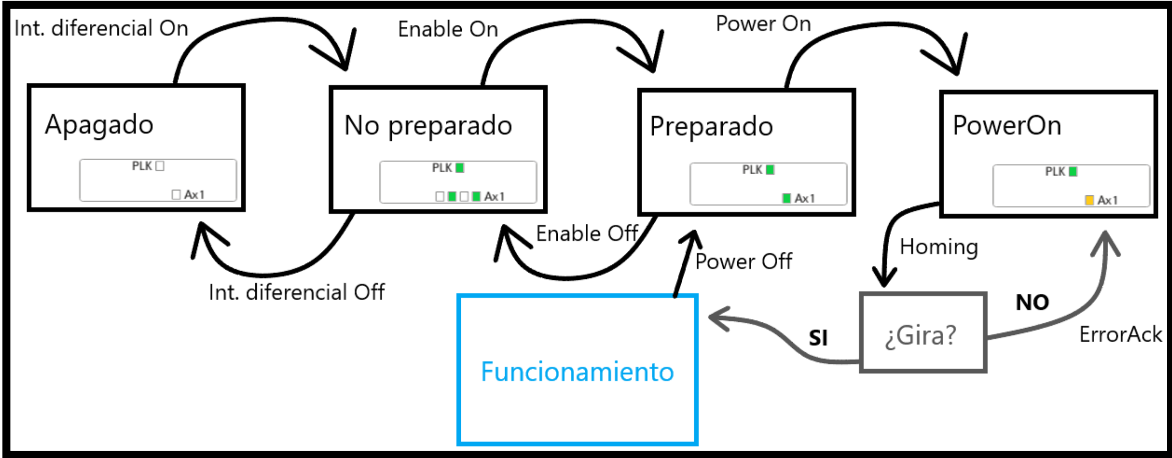


Ilustración 23. Cambio de estados y señalización en el servoaccionamiento. Muestra el flujo entre estados de un servoaccionamiento durante su encendido.

Cabe mencionar que la imagen no muestra todos los casos posibles de pasos entre las fases mencionadas, y se fija principalmente en los indicadores del ACOPOS P3. El proceso que se menciona a continuación hace referencia al encendido de un motor particular y en caso de querer poner ambos en funcionamiento se debe realizar para ambos.

8.4.1. Puesta en marcha

Para la puesta en marcha se parte desde un estado de apagado, donde la corriente no entra al banco de motores y, por tanto, no enciende ninguno de los interruptores del ACOPOS P3. Al subir el interruptor diferencial y permitir el paso de corriente al sistema, tras dejarlo unos segundos para que el programa empiece a funcionar y se sincronicen las funciones, se llega a un estado de no preparado.

En el estado de no preparado el programa cíclico se está ejecutando, pero el servomotor todavía no ha recibido la señal de enable, por lo que no está permitido su funcionamiento. En este estado el ACOPOS P3 tendrá el display del PLK en verde fijo y el Ax1 en verde parpadeando. Para avanzar al siguiente estado hay que pulsar el botón enable, permitiendo así el funcionamiento del motor.

Este estado es el estado preparado, en este el servomotor ya permite el funcionamiento del motor. Este estado se muestra con el PLK en verde fijo y el Ax1 en verde fijo. El siguiente paso es permitir el suministro de potencia al motor, para ello se hace uso del botón Power. En este caso se pasa a un estado donde la potencia ya llega al eje.

Una vez en el estado que se ha llamado PowerOn, falta hacer uso de la función Homing que hace que el motor reconozca al servomotor y se permita el funcionamiento. Al hacer uso del Homing conviene probar su funcionamiento. En caso de encontrarse funcionando, el motor se habría

iniciado correctamente. En caso de no darse este funcionamiento, significa que el módulo se encuentra en una fase de error y primero se debe reconocer dicho error.

Que el motor se encuentre en una fase de error, puede venir dado por haber intentado ejecutar una orden (ej. Start) sin haber llegado a la etapa de funcionamiento. Para poder seguir con el funcionamiento, se debe hacer uso del botón error acknowledge. Este botón indica al motor que se es consciente de que ha generado un error en el motor y que a la vista de las condiciones externas el sistema puede seguir funcionando de forma segura. Una vez usado dicho botón se debe realizar uso de nuevo del Homing para comenzar a usar el motor con normalidad. (Ver 6.4.3. Recomendaciones)

8.4.2. Apagado

Para apagar el motor de forma adecuada, el primer paso es pararlo de forma segura. Una vez completamente parado, se debe desconectar la entrada de potencia pulsando de nuevo el botón de power. Tras esto se debe quitar el enable (pulsando el botón de alarma) de dicho servomotor y para finalizar desconectar la corriente con el interruptor diferencial. De esta forma, el motor no se involucra en movimientos descontrolados y se evita que para el próximo uso el servomotor parta de un estado de error.

A pesar de que esa sea la forma óptima de desconectar el motor, cabe mencionar que existe otra forma de apagar el motor para casos extremos. Esto es realizando la desconexión con el botón de alarma, en caso de estar en movimiento los motores, dicho botón desconecta los enables de ambos motores. Se debe tener en cuenta que este movimiento no para el motor, únicamente corta el suministro de energía al mismo, por lo que seguirá girando libre hasta que se pare por efectos del rozamiento. Se ruega no usar esta forma de desconexión a no ser que se trate de una emergencia ya que esto puede reducir la vida útil de los elementos del banco.

8.4.3. Inconvenientes y recomendaciones

Existe un claro inconveniente durante el arranque. No tener la certeza de si uno de los módulos de motores se encuentra en fase de error. Esto no supone un gran inconveniente en el caso del primer motor, ya que basta con aplicarle una velocidad para ver si se encuentra en funcionamiento o no. Sin embargo, el segundo motor al solo tener la función de aplicar par resistente, no hay una forma evidente de saber si está en fase de funcionamiento o se trata de un error.

Para asegurar que dicho segundo motor se encuentra en funcionamiento se recomienda hacer uso del botón de error acknowledge como un paso más sistemático. De esta forma en caso de existir un error previo pasará de dicha fase, y en caso de no existir ningún error se mantendrá así.

8.5. Graficado

Este último apartado hace referencia al uso de la función de graficado de velocidad disponible en el segundo fragmento de la aplicación. Para que sea útil el uso de dicha función, es necesario que

la aplicación tenga una conexión ya establecida con el banco de motores (7.3. Conexión App – Banco de ensayos). En cuanto se de esta conexión comienza un flujo de datos continuo que actualiza constantemente el valor de velocidad real del motor.

Para comenzar el plasmado de los datos en la gráfica basta con usar el botón de play. De esta forma comenzará una gráfica móvil que quita de la vista viejos datos al ritmo al que se van introduciendo nuevos.

En caso de querer parar el plasmado de datos se debe hacer uso del botón pause. Esta función es de mucha utilidad para ver los datos plasmados ante un cambio de par o de velocidad. En caso de querer visualizar los datos que se han dejado atrás, la gráfica puede moverse deslizando el dedo sobre la gráfica. Además, si se quiere ver los datos con más detalle se puede hacer zoom deslizando dos dedos sobre la gráfica hacia extremos contrarios de la misma.

9. CONCLUSIÓN Y PROPUESTAS DE MEJORA

El objetivo del trabajo era el desarrollo de un banco de motores mediante el control por una aplicación. Principalmente el trabajo se ha abordado en tres partes principales: montaje físico, aplicación android y programa en Automation Studio. Los tres módulos se han desarrollado satisfactoriamente logrando los siguientes objetivos:

- Aplicación intuitiva para el manejo remoto de ambos motores unidos por su eje. Permitiendo la variación de parámetros de funcionamiento de los motores y su manejo básico.
- Conexión mediante método TCP/IP mediante la creación de una red mediante un router, que permite la conexión al servidor creado por el PLC. Dando la posibilidad de conexión y desconexión de la red.
- Obtención de datos de velocidad a tiempo real, y con una respuesta adecuada para los objetivos de funcionamiento que se espera requerir.
- Graficado de datos de velocidad mediante una gráfica interactiva que permite funciones como visualizar parar la gráfica, visualizar datos previos o hacer zoom a la misma.

Se considera que el proyecto realizado cumple las funciones básicas requeridas, sin embargo, durante la realización de este se han observado múltiples opciones de mejora que pueden dar paso a trabajos futuros. Estos están dirigidos a la mejora global de este proyecto como a la ampliación de propuestas para trabajar con este. Las propuestas son las siguientes:

- Obtención de datos de par real para el motor 1 y establecimiento de una nueva pestaña en el TabView de la aplicación para mostrar dichos datos.
- Pantalla de muestra de estado de ambos motores, reduciendo así la dependencia del sistema de displays de los servoaccionamientos a la hora de conocer el estado de los mismos. Así se solucionaría el problema mencionado de desconocimiento del estado de error (6.4.3. Inconvenientes y recomendaciones).
- Desarrollo de otras funcionalidades del motor como desplazamiento a posición x y cambio progresivo de velocidad entre otros.
- Preparación de un cuadernillo de prácticas con posibles ensayos sobre este proyecto, donde se aclaren conceptos teóricos complejos mediante los mismos. Ej. Ensayos de vacío, Estudio de la variación de la curva de velocidad a distintos pares resistentes u otros.

10. BIBLIOGRAFÍA

Edición de B&R. “Guía básica de Motion ACP10. Automation Studio AS4.3” [2019].

Editores de B&R. Guía de ayuda general de Automation Studio. Uso exclusivo desde software.

Edición de B&R. “8LS...-3 three-phase synchronous motors. User's manual.” Version: 2.50 [2018].

Edición de B&R. “ACOPOS P3. User's manual” Version: 1.10 [2017].

Vipin, creador de CODERZHEAVEN . Código sobre Servidores y Clientes TCP [en línea]. Página web coderzheaven.com, 2020 [fecha de consulta: 7 de mayo del 2020]. Disponible en <http://www.coderzheaven.com/2017/05/01/client-server-programming-in-android-send-message-to-the-client-and-back/> .

Editores de androidwave. Código sobre Comunicación entre fragmentos con el uso de viewModel [en línea]. Página web androidwave.com, 2020 [fecha de consulta: 18 de mayo del 2020]. Disponible en < https://androidwave.com/fragment-communication-using-viewmodel/> .

Jjoe64, usuario de GitHub creador de la apliación para el uso de GraphView en Android studio. Código sobre uso de graphView para graficado de funciones [en línea]. Página web github.com 2020 [fecha de consulta: 27 de mayo del 2020]. Disponible en <https://github.com/jjoe64/GraphView/wiki> .

Editores de la página material.io. Descarga de iconos para la parte visual de la aplicación [en línea]. Página web material.io 2020 [fecha de consulta: 2 de mayo del 2020]. Disponible en <https://material.io/resources/icons/> .

Editores de la página de desarrolladores de Android. Consultas varias de información para código y para memoria. Página web developer.android.com, 2020 [fecha de consulta: meses de mayo y junio del 2020]. Enlaces usados:
<https://developer.android.com/topic/libraries/architecture/viewmodel.html#java>

<https://developer.android.com/topic/libraries/architecture/adding-components>

<https://developer.android.com/topic/libraries/architecture/viewmodel>

<https://developer.android.com/training/basics/fragments/pass-data-between>

<https://developer.android.com/guide/platform?hl=es-419>

<https://developer.android.com/studio/build/dependencies?hl=es-419>

<https://developer.android.com/guide/components/activities/intro-activities?hl=es>

<https://developer.android.com/guide/components/fragments?hl=es>

Chris usuario de StackOverflow. Código sobre programa servidor y la función DataInputStream [en línea]. Página web stackocerflow.com 2020 [fecha de consulta: 9 de mayo del 2020]. Disponible en < https://stackoverflow.com/questions/28187038/tcp-client-server-program-datainputstream-dataoutputstream-issue> .

Oscar Avelia, editor de abb. Newsletter sobre reducción de armónicos [en línea]. Página web stackocerflow.com 2020 [fecha de consulta: 19 de junio del 2020]. Disponible en <

https://new.abb.com/docs/librariesprovider78/newsletters/actualidad-colombia/actualidad-413.pdf?sfvrsn=2> .

Editores de HarmonicDrive. Información básica sobre servomotores [en linea]. Página web harmonicdrive.de 2020 [fecha de consulta: 20 de junio de 2020]. Disponible en <https://harmonicdrive.de/es/glosario/servomotor>

Editores de xakataandroid. Información sobre Android para la memoria [en linea]. Página web xakatandroid.com 2020 [fecha de consulta: 22 de junio de 2020]. Disponible en <https://www.xatakandroid.com/sistema-operativo/que-es-android>

Editores de B&R Automation. Múltiples consultas sobre información de producto [en linea]. Página web br-automation.com, 2020 [fecha de consulta: Repeditas veces entre los meses de febrero y junio del 2020]. Disponible en < https://www.br-automation.com/ >

TRABAJO FINAL DE GRADO

Valencia. 2019/2020 Iván Arenas Kelbelova

PRESUPUESTO

DISEÑO Y DESARROLLO DE UN BANCO DE PRUEBAS DE SERVOACCIONAMIENTOS DE 4,5kW CON OPERACIÓN REMOTA

Indice de Presupuesto

1. Introducción 1

2. Precios de los jornales 1

3. Precios de los materiales 1

4. Precios unitarios 2

5. Precios descompuestos 3

6. Presupuesto de ejecución por contrata y licitación 5

1. Introducción

En este documento se estima el presupuesto de ejecución material del montaje de un banco de ensayo de dos servomotores síncronos enfrentados frontalmente. Para afrontarlo adecuadamente e intentando lograr un resultado ajustado a la realidad, esta tarea se ha dividido en varias partes. Se han definido 4 cuadros de precios pertinentes para esta función:

- Cuadro Nº1. Precios de los jornales.
- Cuadro Nº2. Precios de los materiales.
- Cuadro Nº3. Precios Unitarios.
- Cuadro Nº4. Precios Descompuestos.

A partir de estos cuadros se ha hallado el presupuesto parcial del proyecto y el presupuesto de ejecución material y por contrata.

2. Precios de los jornales

El siguiente cuadro muestra los precios de los jornales de las personas involucradas en el proyecto.

Concepto	Descripción	Unidad	Coste
Supervisor del proyecto	Hora de trabajo de docente universitario en el área de ingeniería eléctrica. Encargado de realizar labores de supervisión.	h	60 €
Técnico de laboratorio	Hora de trabajo de técnico de laboratorio. Encargado de realizar labores de apoyo en el montaje físico del banco.	h	30 €
Ingeniero Junior	Hora de trabajo de ingeniero junior. Encargado principal del desarrollo del proyecto, involucrado en distintas funciones.	h	25 €

Tabla 3. Tabla de precios de los jornales.

3. Precios de los materiales

El siguiente cuadro muestra los precios de unidad de material utilizado en el proyecto.

Concepto	Descripción	Unidad	Amortización	Coste
Software Automation Studio	Licencia de uso del software de programación del PLC B&R, Automation Studio.	día	400 días	0.17 €
Software Android Studio	Programa de uso gratuito, Android Studio.	-	-	0 €

Servomotor	Servomotor síncrono Brussles de cuatro polos. Marca B&R clase 8LSA.	ud	-	1275,21 €
Accionamiento trifásico	Servoaccionamiento trifásico, marca B&R tipo ACOPOS P3	ud	-	903,83 €
Filtro de red	Filtro de red marca B&R modelo 8B0F0160H000.A00-1	ud	-	82,23 €
Cable motor	Cable híbrido para servo-accionamiento ACOPOS P3.	ud	-	203,55 €
PLC	PLC marca B&R modelo X20CP1382	ud	-	886,98 €
Router Wifi	Router	ud	-	15 €
PC	Ordenador de mesa completo unidad (torre+pantalla).	meses	5 años	10 €

Tabla 4. Tabla de precios de los materiales.

En la tabla se muestran los precios de los distintos elementos que han sido necesarios para realizar el proyecto junto con una descripción detallada, unidades, tiempo de amortización y precio. Cabe aclarar la amortización dada sobre el uso del PC. Se le ha dado una vida media de 5 años y un precio de 600 €, como la unidad se da en meses, el precio por uso mensual es de 10€.

Como es normal en todos los proyectos hay ciertos elementos que han sido usados puntualmente pero que es muy complicado imputar su uso. Estos se añadirán en forma de costes directos más adelante. Cabe indicar que al comprar los productos en bloque se consiguió una rebaja en todos los productos de B&R de un 32,60%, que ha sido incluida como precio final en la tabla.

4. Precios unitarios

A continuación, se muestra la tabla de precios unitarios de obra para las 4 unidades de obra definidas, con una descripción detallada de cada una de ellas, precio y unidades.

Concepto	**Descripción**	**Unidad**	**Coste**
Montaje Físico	Montaje físico del banco de ensayos de servomotores enfrentados frontalmente. Incluyendo protecciones y requerimientos eléctricos para su funcionamiento idóneo.	Ud	6.439,92 €
Programación PLC	Programación del programa de control del PLC X20CP1382. Programa realizado mediante el software de programación Automation Studio de B&R.	ud	4.008,80 €
Aplicación Android	Programación de la aplicación Android para manejo de motores y recepción de datos de forma remota.	ud	3.916,80 €
Redacción de documentos técnicos	Redacción de documentos técnicos (Memoria y Presupuesto) del proyecto. Se incluye tanto primera redacción como costes de revisión y corrección de esta.	ud	2.152,20 €

Tabla 5. Tabla de precios unitarios.

Estos precios para cada unidad de obra del proyecto aparecen desglosados en el siguiente apartado, especificando los gastos que llevan a la estimación de dicho precio.

5. Precios descompuestos

A continuación, se muestra un listado donde aparecen estimados los recursos necesarios de las tablas anteriores para justificar finalmente el coste del proyecto realizado.

Nº Orden	Descripción de la unidades de obra
U.O.1.	Montaje físico del banco de ensayos de servomotores enfrentados frontalmente. Incluyendo protecciones y requerimientos eléctricos para su funcionamiento idóneo.

COSTES DIRECTOS			
Rendimientos	Descripción	Precio	Importe
20	h. Técnico de laboratorio	30 €	600,00 €
10	h. Ingeniero junior	25 €	250,00 €
2	ud. Servomotor 8LSA	1.275,21 €	2.550,42 €
2	ud. Accionamiento trifásico, ACOPOS P3	903,83 €	1.807,60 €
1	ud. Filtro de red	82,23 €	82,23 €
1	PLC X20CP1382	886,98 €	886,98 €
1	Router	15 €	15,00 €
	4% Costes Directos Complementarios		247,69 €
COSTE TOTAL UNIDAD DE OBRA			6.439,92 €

Tabla 6. Tabla de Unidad de obra 1.

Nº Orden	Descripción de la unidades de obra
U.O.2.	Programación del programa de control del PLC X20CP1382. Programa realizado mediante el software de programación Automation Studio de B&R.

COSTES DIRECTOS			
Rendimientos	Descripción	Precio	Importe
15	h. Supervisor del proyecto	60 €	900,00 €
120	h. Ingeniero junior	25 €	3.000,00 €
60	día. Automation Studio	0,17 €	10,02 €

2	mes. Uso de PC	10 €	20,00 €
	2% Costes Directos Complentarios		78,60 €
COSTE TOTAL UNIDAD DE OBRA			4.008,80 €

Tabla 7. Tabla de unidad de obra 2.

Nº Orden	Descripción de la unidades de obra
U.O.3.	Programación de la aplicación Android para manejo de motores y recepción de datos de forma remota.

COSTES DIRECTOS

Rendimientos	Descripción	Precio	Importe
18	h. Supervisor del proyecto	60 €	1.080,00 €
110	h. Ingeniero junior	25 €	2.750,00 €
60	día. Android Studio	0 €	0 €
1	mes. Uso de PC	10 €	10,00 €
	2% Costes Directos Complentarios		76,8 €
COSTE TOTAL UNIDAD DE OBRA			3.916,80 €

Tabla 8. Tabla de unidad de obra 3.

Nº Orden	Descripción de la unidades de obra
U.O.4.	Redacción de documentos técnicos (Memoria y Presupuesto) del proyecto. Se incluye tanto primera redacción como costes de revisión y corrección de esta.

COSTES DIRECTOS

Rendimientos	Descripción	Precio	Importe
10	h. Supervisor del proyecto	60 €	600,00 €
60	h. Ingeniero junior	25 €	1.500,00 €
60	día. Android Studio	0 €	0 €
1	mes. Uso de PC	10 €	10,00 €
	2% Costes Directos Complentarios		42,2 €
COSTE TOTAL UNIDAD DE OBRA			2.152,20 €

Tabla 9. Tabla de unidad de obra 4.

Cabe mencionar que, en el caso de la primera unidad de obra se imputan mayor porcentaje de costes directos complementarios ya que en esta se han usado numerosas herramientas del laboratorio para realizar las conexiones, cables para las mismas y otros elementos que no se han podido imputar directamente.

6. Presupuesto de ejecución material

En la siguiente tabla se halla el presupuesto de ejecución material.

Nº Unidad	Unidades necesarias	Precio	Importe
U.O.1.	1 ud	6.439,92 €	6.439,92 €
U.O.2.	1 ud	4.008,80 €	4.008,80 €
U.O.3.	1 ud	3.916,80 €	3.916,80 €
U.O.4.	1 ud	2.152,20 €	2.152,20 €
		Presupuesto de ejecución material	**16517,72€**

Tabla 10. Tabla de presupuesto de ejecución material.

6. Presupuesto de ejecución por contrata y licitación

A continuación, se muestra el presupuesto de ejecución por contrata y el presupuesto de licitación.

PRESUPUESTO DE EJECUCIÓN MATERIAL	16.517,72 €
20% GASTOS GENERALES	3.303,54 €
6% BENEFICIO INDUSTRIAL	991,06 €
PRESUPUESTO DE EJECUCIÓN POR CONTRATA	**20.812,32 €**
21% IVA	4.370,59 €
PRESUPUESTO BASE DE LICITACIÓN	**25.182,91 €**

Tabla 11. Tabla de presupuesto de ejecución por contrata y licitación.

Asciende el presente presupuesto a la expresada cantidad de:

VEINTICINCO MIL CIENTO OCHENTA Y DOS EUROS CON NOVENTA Y UN CÉNTIMOS.

ANEXO 1

CÓDIGO DE LA APLICACIÓN REALIZADA EN ANDROID STUDIO.

En este documento se adjunta el código desarrollado en Android studio para la creación de la app móvil, se muestran los módulos principales y evitándose los módulos en los que apenas no se ha tenido aportación.

En las siguientes páginas se van a mostrar las dos clases, el programa MainActivity y los dos fragmentos en los que se basa el proyecto. Estos son VisualDataFragment, StartClient, MainActivity, InitFragment y SharedViewModel.

1. VisualDataFragment

```java
package com.arenaskelbelova.comunicaciontcp.Fragments;
import android.annotation.SuppressLint;
import android.os.Bundle;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.Observer;
import androidx.lifecycle.ViewModelProvider;
import android.os.Handler;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageButton;
import com.arenaskelbelova.comunicaciontcp.R;
import com.jjoe64.graphview.GraphView;
import com.jjoe64.graphview.LegendRenderer;
import com.jjoe64.graphview.series.DataPoint;
import com.jjoe64.graphview.series.LineGraphSeries;

/**
 * A simple {@link Fragment} subclass.
 * Use the {@link VisualDataFragment#newInstance} factory method to
 * create an instance of this fragment.
 */
public class VisualDataFragment extends Fragment implements
View.OnClickListener {
    // TODO: Rename parameter arguments, choose names that match
    // the fragment initialization parameters, e.g. ARG_ITEM_NUMBER
    private static final String ARG_PARAM1 = "param1";
    private static final String ARG_PARAM2 = "param2";
    private static final int TIME_RANGE = 10; // In sec
    private static final int SAMPLE_TIME = 5; // In ms

    // TODO: Rename and change types of parameters
    private int nextPoint = 0;
    private boolean runActivated = false;
    private LineGraphSeries<DataPoint> series;
    private GraphView velChart;
    private float time = 0;
    private Handler plotDataHandler = new Handler();



    public VisualDataFragment() {
        // Required empty public constructor
    }
```

```java
    /**
     * Use this factory method to create a new instance of
     * this fragment using the provided parameters.
     *
     * @param param1 Parameter 1.
     * @param param2 Parameter 2.
     * @return A new instance of fragment VisualDataFragment.
     */
    // TODO: Rename and change types and number of parameters
    public static VisualDataFragment newInstance(String param1, String param2)
{
        VisualDataFragment fragment = new VisualDataFragment();
        Bundle args = new Bundle();
        args.putString(ARG_PARAM1, param1);
        args.putString(ARG_PARAM2, param2);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mParam1 = getArguments().getString(ARG_PARAM1);
            mParam2 = getArguments().getString(ARG_PARAM2);
        }
    }

    @SuppressLint("ResourceType")
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        final View view;
        view = inflater.inflate(R.layout.fragment_visual_data, container,
false);

        SharedViewModel viewModel = new
ViewModelProvider(requireActivity()).get(SharedViewModel.class);
        viewModel.getNewValue().observe(getViewLifecycleOwner(), new
Observer<Integer>() {
            @Override
            public void onChanged(Integer integer) {
                nextPoint = integer;
            }
        });

        ImageButton startChart;
        ImageButton stopChart;
        ImageButton cleanChart;



        velChart = view.findViewById(R.id.velChart);
        startChart = view.findViewById(R.id.btn_start_chart);
        stopChart = view.findViewById(R.id.btn_pause_chart);
        cleanChart = view.findViewById(R.id.btn_delete_chart);
```

```java
        startChart.setOnClickListener(this);
        stopChart.setOnClickListener(this);
        cleanChart.setOnClickListener(this);

        series = new LineGraphSeries<>(new DataPoint[]{
        });
        velChart.addSeries(series);

        velChart.getViewport().setMinX(0);
        velChart.getViewport().setMaxX(TIME_RANGE);


velChart.setBackgroundColor(getResources().getColor(android.R.color.white));
        velChart.setTitle("rpm/s");
        velChart.getViewport().setScalable(true);
        velChart.getViewport().setScrollable(true);

        series.setTitle("Velocidad");
        velChart.getLegendRenderer().setVisible(true);
        velChart.getLegendRenderer().setAlign(LegendRenderer.LegendAlign.TOP);

        return view;
    }


    @Override
    public void onClick(View v) {
        switch (v.getId()){
            case R.id.btn_start_chart:
                runActivated = true;
                plotDataHandler.postDelayed(plotData, SAMPLE_TIME);
                break;
            case R.id.btn_pause_chart:
                runActivated = false;
                break;
            case R.id.btn_delete_chart:
                runActivated = false;
                series.resetData(new DataPoint[]{});
                time = 0;
                velChart.getViewport().setMinX(0);
                velChart.getViewport().setMaxX(TIME_RANGE);
                break;

    }}

    private Runnable plotData = new Runnable() {
        @Override
        public void run() {
            if (runActivated){
                plotDataHandler.postDelayed(plotData, SAMPLE_TIME);

                series.appendData(new DataPoint(time, nextPoint), true, 100);
                velChart.getViewport().setMinY(0);

                if (time<=TIME_RANGE){
                    velChart.getViewport().setMinX(0);
                    velChart.getViewport().setMaxX(TIME_RANGE);
                    velChart.getViewport().setXAxisBoundsManual(true);
                }
```

```
                time += (float)SAMPLE_TIME/1000;

            }
        }
    };


}
```

2. *Clase StartClient*

```
package com.arenaskelbelova.comunicaciontcp.TCPprocess;
import com.arenaskelbelova.comunicaciontcp.Fragments.InitFragment;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;


public class StartClient implements  Runnable {
    private Socket socket;
    private final String IP ="192.168.1.149";
    private final int PORT = 23;

    //Runnable

    @Override
    public void run() {

        try {
            socket = new Socket(IP, PORT);
            // Enters in a while loop until the connection is interrupted
            while (!Thread.currentThread().isInterrupted()) {
                DataInputStream inputStream = new
DataInputStream(socket.getInputStream());
                int message = inputStream.readInt();
                InitFragment.getInstance().showMessage(message);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }


    }


    public void sendMessage(final int message) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    if (null != socket) {
```

```
                        DataOutputStream outputStream = new
DataOutputStream(socket.getOutputStream());
                        outputStream.write(message);

                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    public void stop() throws IOException {
        socket.close();
    }

    public boolean isClosed(){
        return socket.isClosed();
    }

}
```

3. *Clase SharedViewModel*

```
package com.arenaskelbelova.comunicaciontcp.Fragments;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.lifecycle.ViewModel;

public class SharedViewModel extends ViewModel {
    private MutableLiveData<Integer> receivedData = new MutableLiveData<>();

    public void setNewValue(Integer
newValue){receivedData.setValue(newValue); }

    public LiveData <Integer> getNewValue() {return receivedData;}
}
```

4. Fragmento InitFragment

```
package com.arenaskelbelova.comunicaciontcp.Fragments;
import android.annotation.SuppressLint;
import android.graphics.Color;
import android.os.Bundle;
import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.fragment.app.Fragment;
import androidx.lifecycle.ViewModelProvider;
import android.os.Handler;
import android.view.LayoutInflater;
import android.view.View;
```

```java
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.ImageButton;
import android.widget.SeekBar;
import android.widget.TextView;
import com.arenaskelbelova.comunicaciontcp.R;
import com.arenaskelbelova.comunicaciontcp.TCPprocess.StartClient;
import java.io.IOException;

/**
 * A simple {@link Fragment} subclass.
 * Use the {@link InitFragment#newInstance} factory method to
 * create an instance of this fragment.
 */
public class InitFragment extends Fragment implements View.OnClickListener {
    // TODO: Rename parameter arguments, choose names that match
    // the fragment initialization parameters, e.g. ARG_ITEM_NUMBER
    private static final String ARG_PARAM1 = "param1";
    private static final String ARG_PARAM2 = "param2";

    // TODO: Rename and change types of parameters
    private String mParam1;
    private String mParam2;
    // Connection variables
    private Thread thread;
    private StartClient startClient;
    private TextView connectionState;
    private Handler connectHandler = new Handler();
    private Handler recieveHandler = new Handler();

    private SharedViewModel viewModel;

    // SeekBar Variables
    private TextView textVProgress;
    private TextView textTProgress;
    private int progressSendV;
    private int progressSendT;
    private int passVValue;
    private int passTValue;

    // In order to use the method from the class
    private static InitFragment instance;
    private int receivedMessage = 0;




    public InitFragment() {
        // Required empty public constructor
    }

    /**
     * Use this factory method to create a new instance of
     * this fragment using the provided parameters.
     *
     * @param param1 Parameter 1.
     * @param param2 Parameter 2.
     * @return A new instance of fragment InitFragment.
```

```java
     */
    // TODO: Rename and change types and number of parameters
    public static InitFragment newInstance(String param1, String param2) {
        InitFragment fragment = new InitFragment();
        Bundle args = new Bundle();
        args.putString(ARG_PARAM1, param1);
        args.putString(ARG_PARAM2, param2);
        fragment.setArguments(args);
        return fragment;
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            mParam1 = getArguments().getString(ARG_PARAM1);
            mParam2 = getArguments().getString(ARG_PARAM2);
        }
    }

    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,@Nullable
ViewGroup container,
                             @Nullable Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        View vista;
        vista = inflater.inflate(R.layout.fragment_init, container, false);

        viewModel = new
ViewModelProvider(requireActivity()).get(SharedViewModel.class);

        instance = this;

        Button btnConnect;
        Button btnDisconnect;
        Button btnEnable1;
        Button btnPower1;
        Button btnHoming1;
        Button btnErrorAck1;
        Button btnEnable2;
        Button btnPower2;
        Button btnHoming2;
        Button btnErrorAck2;
        ImageButton btnPlay;
        ImageButton btnStop;
        ImageButton btnDisable1;
        ImageButton btnDisable2;
        SeekBar seekBarVelocity;
        SeekBar seekBarTorque;


        btnConnect = vista.findViewById(R.id.btn_connect);
        btnDisconnect = vista.findViewById(R.id.btn_disconnect);
        connectionState = vista.findViewById(R.id.textConnection);
        btnEnable1 = vista.findViewById(R.id.btn_enable_1);
        btnHoming1 = vista.findViewById(R.id.btn_homing_1);
        btnPower1 = vista.findViewById(R.id.btn_power_1);
        btnErrorAck1 = vista.findViewById(R.id.btn_ErrorAck_1);
        btnEnable2 = vista.findViewById(R.id.btn_enable_2);
```

```java
        btnHoming2 = vista.findViewById(R.id.btn_homing_2);
        btnPower2 = vista.findViewById(R.id.btn_power_2);
        btnErrorAck2 = vista.findViewById(R.id.btn_ErrorAck_2);
        btnPlay = vista.findViewById(R.id.btn_play);
        btnStop = vista.findViewById(R.id.btn_stop);
        btnDisable1 = vista.findViewById(R.id.btn_not_enable1);
        btnDisable2 = vista.findViewById(R.id.btn_not_enable2);
        textVProgress = vista.findViewById(R.id.textVProgress);
        textTProgress = vista.findViewById(R.id.textTProgress);
        seekBarVelocity = vista.findViewById(R.id.seekBarVelocity);
        seekBarTorque = vista.findViewById(R.id.seekBarTorque);


        btnConnect.setOnClickListener(this);
        btnDisconnect.setOnClickListener(this);
        btnEnable1.setOnClickListener(this);
        btnPower1.setOnClickListener(this);
        btnHoming1.setOnClickListener(this);
        btnErrorAck1.setOnClickListener(this);
        btnEnable2.setOnClickListener(this);
        btnPower2.setOnClickListener(this);
        btnHoming2.setOnClickListener(this);
        btnErrorAck2.setOnClickListener(this);
        btnPlay.setOnClickListener(this);
        btnStop.setOnClickListener(this);
        btnDisable1.setOnClickListener(this);
        btnDisable2.setOnClickListener(this);


        startClient = new StartClient();






        // Review max values, review communication method
        seekBarVelocity.setOnSeekBarChangeListener(new
SeekBar.OnSeekBarChangeListener() {
            @SuppressLint("SetTextI18n")
            @Override
            public void onProgressChanged(SeekBar seekBar, int progress,
boolean fromUser) {
                passVValue = progress*2000/100;
                int progressSendV = progress;
                textVProgress.setText(""+ passVValue + " rpm");
            }

            @Override
            public void onStartTrackingTouch(SeekBar seekBar) {

            }

            @Override
            public void onStopTrackingTouch(SeekBar seekBar) {
                startClient.sendMessage(progressSendV+);
            }
        });
```

```java
        seekBarTorque.setOnSeekBarChangeListener(new
SeekBar.OnSeekBarChangeListener() {
            @SuppressLint("SetTextI18n")
            @Override
            public void onProgressChanged(SeekBar seekBar, int progress,
boolean fromUser) {
                passTValue = progress*1000/100; // Change the value to the
actual max
                progressSendT= progress+32768;
                textTProgress.setText(""+ passTValue + " N/m");
            }

            @Override
            public void onStartTrackingTouch(SeekBar seekBar) {

            }

            @Override
            public void onStopTrackingTouch(SeekBar seekBar) {
                startClient.sendMessage(progressSendT);
            }
        });


        return vista;
    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.btn_connect:
                // When we start a thread in a Runnable object we are executing
"run()"
                thread = new Thread(startClient);
                thread.start();
                connectHandler.postDelayed(stateConnection, 10);
                //connectHandler.postDelayed(plotData, 5000);

                break;
            case R.id.btn_disconnect:
                try {
                    startClient.stop();
                } catch (IOException e) {
                    e.printStackTrace();
                }


                // Motor 1
            case R.id.btn_play:
                startClient.sendMessage(12);

                break;
            case R.id.btn_stop:
                startClient.sendMessage(11);

                break;
            case R.id.btn_not_enable1:
                startClient.sendMessage(3);
```

```java
                break;
            case R.id.btn_enable_1:
                startClient.sendMessage(1);

                break;
            case R.id.btn_power_1:
                startClient.sendMessage(5);

                break;
            case R.id.btn_homing_1:
                startClient.sendMessage(7);

                break;
            case R.id.btn_ErrorAck_1:
                startClient.sendMessage(9);

                break;

             // Motor 2
            case R.id.btn_enable_2:
                startClient.sendMessage(2);

                break;
            case R.id.btn_power_2:
                startClient.sendMessage(6);

                break;
            case R.id.btn_homing_2:
                startClient.sendMessage(8);

                break;
            case R.id.btn_ErrorAck_2:
                startClient.sendMessage(10);

                break;

            case R.id.btn_not_enable2:
                startClient.sendMessage(4);

                break;
        }
    }

    private Runnable stateConnection = new Runnable() {
        @SuppressLint({"ResourceAsColor", "SetTextI18n"})
        @Override
        public void run() {
            boolean isClosed = startClient.isClosed();
            if (isClosed||!thread.isAlive()) {
                connectionState.setText("Disconnected");
                connectionState.setTextColor(Color.RED);
            } else {
                connectionState.setText("Connected");
                connectionState.setTextColor(Color.GREEN);
            }
        connectHandler.postDelayed(stateConnection, 10);
        };

    };
```

```java
    public static InitFragment getInstance() {
        return instance;
    }

    public void showMessage(final int message) {
        recieveHandler.post(new Runnable() {
            @Override
            public void run() {
                if (receivedMessage != message){
                receivedMessage = message;
                viewModel.setNewValue(message);
                }

            }
        });
    }


}
```

5. MainActivity

```java
package com.arenaskelbelova.comunicaciontcp;

import android.os.Bundle;

import com.google.android.material.floatingactionbutton.FloatingActionButton;
import com.google.android.material.snackbar.Snackbar;
import com.google.android.material.tabs.TabLayout;

import androidx.viewpager.widget.ViewPager;
import androidx.appcompat.app.AppCompatActivity;

import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.Button;

import com.arenaskelbelova.comunicaciontcp.ui.main.SectionsPagerAdapter;

public class MainActivity extends AppCompatActivity  {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        SectionsPagerAdapter sectionsPagerAdapter = new
SectionsPagerAdapter(this, getSupportFragmentManager());
        ViewPager viewPager = findViewById(R.id.view_pager);
        viewPager.setAdapter(sectionsPagerAdapter);
        TabLayout tabs = findViewById(R.id.tabs);
        tabs.setupWithViewPager(viewPager);

    }
}
```

ANEXO 2

CÓDIGO DE LA APLICACIÓN REALIZADA EN AUTOMATION STUDIO.

En este Segundo anexo se muestra el código de los cuatro módulos principales de la aplicación de control Motion1, Motion2, Program y Server. En estos se muestra el código de inicialización y el bucle de funcionamiento.

1. Módulo Motion1

```
PROGRAM _INIT

(* Axis reference: Unicamente refiere una variable interna tipo UDINT eje físico "gAxis01"
        con la intención de referirse a el por codigo *)
Axis1Obj := ADR(gAxis01);

AxisStep := STATE_WAIT; (* start step *)
        (*Inicialiazación de parámetros físicos. Posteriormente podremos cambiarlos por
        codigo en los casos que nos interese*)
BasicControl.Parameter.Velocity             := 1000;  (*velocity for movement*)
BasicControl.Parameter.Acceleration         := 5000;  (*acceleration for movement*)
BasicControl.Parameter.Deceleration         := 5000;  (*deceleration for movement*)
BasicControl.Parameter.JogVelocity          := 400;   (*velocity for jogging *)
END_PROGRAM


PROGRAM _CYCLIC

(****************************************************************
        Control Sequence
        Unicamente chequea si todo está correcto para iniciar el motor.
****************************************************************)
(* status information is read before the step sequencer to attain a shorter reaction time
*)
        (*Almacena en su propia estructura de variables "Basicontrol" los distintos estados
leidos de la situación del eje*)
        (*Además va dando valores a parámetros de bloques de funciones que luego influiran
en el uso*)

(*INICIO DE FASE 1/3 LCTURA DE ESTADO DEL MOTOR*)

(*********************** MC_READSTATUS ***********************)
        MC_ReadStatus_0.Enable := NOT(MC_ReadStatus_0.Error);
        MC_ReadStatus_0.Axis := Axis1Obj;
        MC_ReadStatus_0();
        BasicControl.AxisState.Disabled             := MC_ReadStatus_0.Disabled;
        BasicControl.AxisState.StandStill           := MC_ReadStatus_0.StandStill;
        BasicControl.AxisState.Stopping             := MC_ReadStatus_0.Stopping;
        BasicControl.AxisState.Homing               := MC_ReadStatus_0.Homing;
        BasicControl.AxisState.DiscreteMotion       := MC_ReadStatus_0.DiscreteMotion;
        BasicControl.AxisState.ContinuousMotion     := MC_ReadStatus_0.ContinuousMotion;
        BasicControl.AxisState.SynchronizedMotion   := MC_ReadStatus_0.SynchronizedMotion;
        BasicControl.AxisState.ErrorStop            := MC_ReadStatus_0.Errorstop;

        (*******************MC_BR_READDRIVESTATUS**********************)
        MC_BR_ReadDriveStatus_0.Enable := NOT(MC_BR_ReadDriveStatus_0.Error);
        MC_BR_ReadDriveStatus_0.Axis := Axis1Obj;
        MC_BR_ReadDriveStatus_0.AdrDriveStatus := ADR(BasicControl.Status.DriveStatus);
        MC_BR_ReadDriveStatus_0();

        (******************* MC_READACTUALPOSITION ********************)
        MC_ReadActualPosition_0.Enable := (NOT(MC_ReadActualPosition_0.Error));
        MC_ReadActualPosition_0.Axis := Axis1Obj;
        MC_ReadActualPosition_0();
        IF(MC_ReadActualPosition_0.Valid = TRUE)THEN
                BasicControl.Status.ActPosition := MC_ReadActualPosition_0.Position;
```

```
	END_IF

	(******************** MC_READACTUALVELOCITY ********************)
	MC_ReadActualVelocity_0.Enable := (NOT(MC_ReadActualVelocity_0.Error));
	MC_ReadActualVelocity_0.Axis := Axis1Obj;
	MC_ReadActualVelocity_0();
	IF(MC_ReadActualVelocity_0.Valid = TRUE)THEN
		BasicControl.Status.ActVelocity := MC_ReadActualVelocity_0.Velocity;
		ActualVelocity := BasicControl.Status.ActVelocity;
	END_IF

	(************************ MC_READAXISERROR *********************)
	MC_ReadAxisError_0.Enable := NOT(MC_ReadAxisError_0.Error);
	MC_ReadAxisError_0.Axis := Axis1Obj;
	MC_ReadAxisError_0.DataAddress := ADR(BasicControl.Status.ErrorText);
	MC_ReadAxisError_0.DataLength := SIZEOF(BasicControl.Status.ErrorText);
	MC_ReadAxisError_0.DataObjectName := 'acp10etxen';
	MC_ReadAxisError_0();

	(**************** CHECK FOR GENERAL AXIS ERROR *****************)
	IF ((MC_ReadAxisError_0.AxisErrorID <> 0) AND (MC_ReadAxisError_0.Valid = TRUE))
THEN
		AxisStep := STATE_ERROR_AXIS;
		(***************** CHECK IF POWER SHOULD BE OFF *******************)
	ELSIF ((BasicControl.Command.Power = FALSE) AND (MC_ReadAxisError_0.Valid = TRUE))
THEN
		IF ((MC_ReadStatus_0.Errorstop = TRUE) AND (MC_ReadStatus_0.Valid = TRUE))
THEN
			AxisStep := STATE_ERROR_RESET;
		ELSE
			AxisStep := STATE_WAIT;
		END_IF
	END_IF

	(* central monitoring OF stop command attains a shorter reaction TIME in CASE OF
emergency stop *)
	(* Controla el comando stop en caso de que sea usado*)
	(*****************CHECK FOR STOP COMMAND***********************)
	IF (BasicControl.Command.Stop = TRUE) THEN
		IF ((AxisStep >= STATE_HOME) AND (AxisStep <= STATE_ERROR)) THEN
			(* reset all FB execute inputs we use *)
			MC_Home_0.Execute := 0;
			MC_Stop_0.Execute := 0;
			MC_MoveAbsolute_0.Execute := 0;
			MC_MoveAdditive_0.Execute := 0;
			MC_MoveVelocity_0.Execute := 0;
			MC_ReadAxisError_0.Acknowledge := 0;
			MC_Reset_0.Execute := 0;
			MC_Halt_0.Execute := 0;

			(* reset user commands *)
			BasicControl.Command.Halt := 0;
			BasicControl.Command.Home := 0;
			BasicControl.Command.MoveJogPos := 0;
			BasicControl.Command.MoveJogNeg := 0;
			BasicControl.Command.MoveVelocity := 0;
			BasicControl.Command.MoveAbsolute := 0;
			BasicControl.Command.MoveAdditive := 0;
			AxisStep := STATE_STOP;
		END_IF
	END_IF
	(* FIN DE LA FASE 1/3 ERRORES ESTADO DE FB*)
```

```
(*INICIO DE FASE 2/3 CASE*)
(*En esta fase se provee de los parametros necesarios para cada acción pero por
claridad no se ejecutan aquí,
se hará en el final del programa. Podíamos decir que es una fase de definición*)
CASE AxisStep OF

        (******************* WAIT ************************)
        STATE_WAIT:  (* STATE: Wait -> AxisStep=0; *) //
                IF (BasicControl.Command.Power = TRUE) THEN
                        AxisStep := STATE_POWER_ON;   // Si hemos dado la orden desde
BC debe entrar en este step.
                ELSE
                        MC_Power_0.Enable := FALSE;
                END_IF

                (* reset all FB execute inputs we use *)
                MC_Home_0.Execute := FALSE;
                MC_Stop_0.Execute := FALSE;
                MC_MoveAbsolute_0.Execute := FALSE;
                MC_MoveAdditive_0.Execute := FALSE;
                MC_MoveVelocity_0.Execute := FALSE;
                MC_Halt_0.Execute := FALSE;
                MC_ReadAxisError_0.Acknowledge := FALSE;
                MC_Reset_0.Execute := FALSE;

                (* reset user commands *)
                BasicControl.Command.Stop := FALSE;
                BasicControl.Command.Halt := FALSE;
                BasicControl.Command.Home := FALSE;
                BasicControl.Command.MoveJogPos := FALSE;
                BasicControl.Command.MoveJogNeg := FALSE;
                BasicControl.Command.MoveVelocity := FALSE;
                BasicControl.Command.MoveAbsolute := FALSE;
                BasicControl.Command.MoveAdditive := FALSE;

                BasicControl.Status.ErrorID := 0;

                (******************* POWER ON *********************)
        STATE_POWER_ON:  (* STATE: Power on -> AxisStep=1 *)
                MC_Power_0.Enable := TRUE;
                IF (MC_Power_0.Status = TRUE) THEN
                        AxisStep := STATE_READY;//Verifica que el motor esté encedido
y nos manda a READY
                END_IF
                (* if a power error occured go to error state *)
                IF (MC_Power_0.Error = TRUE) THEN
                        BasicControl.Status.ErrorID := MC_Power_0.ErrorID;
                        AxisStep := STATE_ERROR;
                END_IF

                (******************* READY *********************)
        STATE_READY:  (* STATE: Waiting for commands ->AxisStep=10 *)

                IF (BasicControl.Command.Home = TRUE)THEN
                        BasicControl.Command.Home := FALSE;
                        AxisStep := STATE_HOME;

                ELSIF (BasicControl.Command.Stop = TRUE) THEN
                        AxisStep := STATE_STOP;

                ELSIF (BasicControl.Command.MoveJogPos = TRUE) THEN
                        AxisStep := STATE_JOG_POSITIVE;

                ELSIF (BasicControl.Command.MoveJogNeg = TRUE) THEN
                        AxisStep := STATE_JOG_NEGATIVE;

                ELSIF (BasicControl.Command.MoveAbsolute = TRUE) THEN
```

```
					BasicControl.Command.MoveAbsolute := FALSE;
					AxisStep := STATE_MOVE_ABSOLUTE;

				ELSIF (BasicControl.Command.MoveAdditive = TRUE) THEN
					BasicControl.Command.MoveAdditive := FALSE;
					AxisStep := STATE_MOVE_ADDITIVE;

				ELSIF (BasicControl.Command.MoveVelocity = TRUE) THEN
					BasicControl.Command.MoveVelocity := FALSE;
					AxisStep := STATE_MOVE_VELOCITY;

				ELSIF (BasicControl.Command.Halt = TRUE) THEN
					BasicControl.Command.Halt := FALSE;
					AxisStep := STATE_HALT;
				ELSIF (BasicControl.Command.PowerOff := TRUE) THEN
					BasicControl.Command.PowerOff := FALSE;
					AxisStep := STATE_POWER_OFF;
				END_IF


				(******************* POWER OFF *********************)
			STATE_POWER_OFF:
				MC_Power_0.Enable := FALSE;
				IF (MC_Power_0.Status = FALSE) THEN
					AxisStep := STATE_WAIT;//Verifica que el motor se apague y
vuelve a la fase wait
				END_IF
				(* gestiona el error *)
				IF (MC_Power_0.Error = TRUE) THEN
					BasicControl.Status.ErrorID := MC_Power_0.ErrorID;
					AxisStep := STATE_ERROR;
				END_IF

				(******************* HOME *********************)
			STATE_HOME:  (* STATE: start homing process -> AxisStep=2*)
				MC_Home_0.Position := BasicControl.Parameter.HomePosition;
				MC_Home_0.HomingMode := BasicControl.Parameter.HomeMode;
				MC_Home_0.Execute := TRUE;
				IF (MC_Home_0.Done = TRUE) THEN
					MC_Home_0.Execute := FALSE;
					AxisStep := STATE_READY;// Se trata de una etapa temporal pasa
automáticamente a la siguiente.
				END_IF
				(* if a homing error occured go to error state *)
				IF (MC_Home_0.Error = TRUE) THEN
					MC_Home_0.Execute := FALSE;
					BasicControl.Status.ErrorID := MC_Home_0.ErrorID;
					AxisStep := STATE_ERROR;
				END_IF

				(*********************HALT_MOVEMENT************************)
			STATE_HALT:   (* STATE: Halt movement *) // Para el motor y vuelve a
STATE_READY para realizar otra acción
				MC_Halt_0.Deceleration := BasicControl.Parameter.Deceleration;
				MC_Halt_0.Execute := TRUE;
				IF (MC_Halt_0.Done = TRUE) THEN
					MC_Halt_0.Execute := FALSE;
					AxisStep := STATE_READY;
				END_IF
				(* check if error occured *)
				IF (MC_Halt_0.Error = TRUE) THEN
					BasicControl.Status.ErrorID := MC_Halt_0.ErrorID;
					MC_Halt_0.Execute := FALSE;
					AxisStep := STATE_ERROR;
				END_IF

				(********************* STOP MOVEMENT ***********************)
```

```
		STATE_STOP:  (* STATE: Stop movement *) // Para el motor hasta que se retire
la acción TRUE
			MC_Stop_0.Deceleration := BasicControl.Parameter.Deceleration;
			MC_Stop_0.Execute := TRUE;
			(* if axis is stopped go to ready state *)
			IF ((MC_Stop_0.Done = TRUE) AND (BasicControl.Command.Stop = FALSE))
THEN
				MC_Stop_0.Execute := FALSE;
				AxisStep := STATE_READY;
			END_IF
			(* check if error occured *)
			IF (MC_Stop_0.Error = TRUE) THEN
				BasicControl.Status.ErrorID := MC_Stop_0.ErrorID;
				MC_Stop_0.Execute := FALSE;
				AxisStep := STATE_ERROR;
			END_IF

			(******************* START JOG MOVEMENT POSITVE
*********************)
		STATE_JOG_POSITIVE:  (* STATE: Start jog movement in positive direction *)
// Su moviemiento para unicamente con STOP o
			// Haciendo BC.Parameter.JV= FALSE de nuevo que hace el equivalente a
un Halt. Halt no actua
			MC_MoveVelocity_0.Velocity      :=
BasicControl.Parameter.JogVelocity;
			MC_MoveVelocity_0.Acceleration  :=
BasicControl.Parameter.Acceleration;
			MC_MoveVelocity_0.Deceleration  :=
BasicControl.Parameter.Deceleration;
			MC_MoveVelocity_0.Direction     := mcPOSITIVE_DIR;
			MC_MoveVelocity_0.Execute := TRUE;
			IF (BasicControl.Command.MoveJogPos = FALSE) THEN
				MC_MoveVelocity_0.Execute := FALSE;
				AxisStep := STATE_HALT;
			END_IF
			(* check if error occured *)
			IF (MC_MoveVelocity_0.Error = TRUE) THEN
				BasicControl.Status.ErrorID := MC_MoveVelocity_0.ErrorID;
				MC_MoveVelocity_0.Execute := FALSE;
				AxisStep := STATE_ERROR;
			END_IF

			(******************* START JOG MOVEMENT NEGATIVE
*********************)
		STATE_JOG_NEGATIVE:  (* STATE: Start jog movement in negative direction *)
			MC_MoveVelocity_0.Velocity      :=
BasicControl.Parameter.JogVelocity;
			MC_MoveVelocity_0.Acceleration  :=
BasicControl.Parameter.Acceleration;
			MC_MoveVelocity_0.Deceleration  :=
BasicControl.Parameter.Deceleration;
			MC_MoveVelocity_0.Direction      := mcNEGATIVE_DIR;
			MC_MoveVelocity_0.Execute := TRUE;
			IF (BasicControl.Command.MoveJogNeg = FALSE) THEN
				MC_MoveVelocity_0.Execute := FALSE;
				AxisStep := STATE_HALT;
			END_IF
			(* check if error occured *)
			IF (MC_MoveVelocity_0.Error = TRUE) THEN
				BasicControl.Status.ErrorID := MC_MoveVelocity_0.ErrorID;
				MC_MoveVelocity_0.Execute := FALSE;
				AxisStep := STATE_ERROR;
			END_IF

			(******************* START ABSOLUTE MOVEMENT *********************)
		STATE_MOVE_ABSOLUTE:  (* STATE: Start absolute movement *)
			MC_MoveAbsolute_0.Position      := BasicControl.Parameter.Position;
```

```
			MC_MoveAbsolute_0.Velocity      := BasicControl.Parameter.Velocity;
			MC_MoveAbsolute_0.Acceleration  :=
BasicControl.Parameter.Acceleration;
			MC_MoveAbsolute_0.Deceleration  :=
BasicControl.Parameter.Deceleration;
			MC_MoveAbsolute_0.Direction     := BasicControl.Parameter.Direction;
			MC_MoveAbsolute_0.Execute := TRUE;
			(* check if commanded position is reached *)
			IF (BasicControl.Command.Halt) THEN
				BasicControl.Command.Halt := FALSE;
				MC_MoveAbsolute_0.Execute := FALSE;
				AxisStep := STATE_HALT;
			ELSIF (MC_MoveAbsolute_0.Done = TRUE) THEN
				MC_MoveAbsolute_0.Execute := FALSE;
				AxisStep := STATE_READY;
			END_IF
			(* check if error occured *)
			IF (MC_MoveAbsolute_0.Error = TRUE) THEN
				BasicControl.Status.ErrorID := MC_MoveAbsolute_0.ErrorID;
				MC_MoveAbsolute_0.Execute := FALSE;
				AxisStep := STATE_ERROR;
			END_IF

			(******************* START ADDITIVE MOVEMENT *********************)
		STATE_MOVE_ADDITIVE:  (* STATE: Start additive movement *)
			MC_MoveAdditive_0.Distance      := BasicControl.Parameter.Distance;
			MC_MoveAdditive_0.Velocity      := BasicControl.Parameter.Velocity;
			MC_MoveAdditive_0.Acceleration  :=
BasicControl.Parameter.Acceleration;
			MC_MoveAdditive_0.Deceleration  :=
BasicControl.Parameter.Deceleration;
			MC_MoveAdditive_0.Execute := TRUE;
			(* check if commanded distance is reached *)
			IF (BasicControl.Command.Halt) THEN
				BasicControl.Command.Halt := FALSE;
				MC_MoveAdditive_0.Execute := FALSE;
				AxisStep := STATE_HALT;
			ELSIF (MC_MoveAdditive_0.Done = TRUE) THEN
				MC_MoveAdditive_0.Execute := FALSE;
				AxisStep := STATE_READY;
			END_IF
			(* check if error occured *)
			IF (MC_MoveAdditive_0.Error = TRUE) THEN
				BasicControl.Status.ErrorID := MC_MoveAdditive_0.ErrorID;
				MC_MoveAdditive_0.Execute := FALSE;
				AxisStep := STATE_ERROR;
			END_IF

			(******************* START VELOCITY MOVEMENT *********************)
		STATE_MOVE_VELOCITY:  (* STATE: Start velocity movement *)
			MC_MoveVelocity_0.Velocity      := BasicControl.Parameter.Velocity;
			MC_MoveVelocity_0.Acceleration  :=
BasicControl.Parameter.Acceleration;
			MC_MoveVelocity_0.Deceleration  :=
BasicControl.Parameter.Deceleration;
			MC_MoveVelocity_0.Direction     := BasicControl.Parameter.Direction;
			MC_MoveVelocity_0.Execute := TRUE;
			(* check if commanded velocity is reached *)
			IF (BasicControl.Command.Halt) THEN
				BasicControl.Command.Halt := FALSE;
				MC_MoveVelocity_0.Execute := FALSE;
				AxisStep := STATE_HALT;
			ELSIF (MC_MoveVelocity_0.InVelocity = TRUE) THEN
				MC_MoveVelocity_0.Execute := FALSE;
				AxisStep := STATE_READY;
			END_IF
			(* check if error occured *)
```

```
				IF (MC_MoveVelocity_0.Error = TRUE) THEN
					BasicControl.Status.ErrorID := MC_MoveVelocity_0.ErrorID;
					MC_MoveVelocity_0.Execute := FALSE;
					AxisStep := STATE_ERROR;
				END_IF

				(******************** FB-ERROR OCCURED *************************)
			STATE_ERROR:  (* STATE: Error *)
				(* check if FB indicates an axis error -> AxisStep=100*)
				IF (MC_ReadAxisError_0.AxisErrorCount<>0) THEN
					AxisStep := STATE_ERROR_AXIS;
				ELSE
					IF (BasicControl.Command.ErrorAcknowledge = TRUE) THEN
						BasicControl.Command.ErrorAcknowledge := FALSE;
						BasicControl.Status.ErrorID := 0;
						(* reset axis if it is in axis state ErrorStop *)
						IF ((MC_ReadStatus_0.Errorstop = TRUE) AND
(MC_ReadStatus_0.Valid = TRUE)) THEN
							AxisStep := STATE_ERROR_RESET;
						ELSE
							AxisStep := STATE_WAIT;
						END_IF
					END_IF
				END_IF

				(******************** AXIS-ERROR OCCURED *************************)
			STATE_ERROR_AXIS:  (* STATE: Axis Error *)
				IF (MC_ReadAxisError_0.Valid = TRUE) THEN
					IF (MC_ReadAxisError_0.AxisErrorID <> 0) THEN
						BasicControl.Status.ErrorID :=
MC_ReadAxisError_0.AxisErrorID;
					END_IF
					MC_ReadAxisError_0.Acknowledge := FALSE;
					IF (BasicControl.Command.ErrorAcknowledge = TRUE) THEN
						BasicControl.Command.ErrorAcknowledge := FALSE;
						(* acknowledge axis error *)
						IF (MC_ReadAxisError_0.AxisErrorID <> 0) THEN
							MC_ReadAxisError_0.Acknowledge := TRUE;
						END_IF
					END_IF
					IF (MC_ReadAxisError_0.AxisErrorCount = 0) THEN
						(* reset axis if it is in axis state ErrorStop *)
						BasicControl.Status.ErrorID := 0;
						IF ((MC_ReadStatus_0.Errorstop = TRUE) AND
(MC_ReadStatus_0.Valid = TRUE)) THEN
							AxisStep := STATE_ERROR_RESET;
						ELSE
							AxisStep := STATE_WAIT;
						END_IF
					END_IF
				END_IF

				(******************** RESET DONE *************************)
			STATE_ERROR_RESET:  (* STATE: Wait for reset done *)
				MC_Reset_0.Execute := TRUE;
				(* reset MC_Power.Enable if this FB is in Error*)
				IF (MC_Power_0.Error = TRUE) THEN
					MC_Power_0.Enable := FALSE;
				END_IF
				IF(MC_Reset_0.Done = TRUE)THEN
					MC_Reset_0.Execute := FALSE;
					AxisStep := STATE_WAIT;
				ELSIF(MC_Reset_0.Error = TRUE) THEN
					MC_Reset_0.Execute := FALSE;
					AxisStep := STATE_ERROR;
				END_IF
			(******************** SEQUENCE END *************************)
```

```
	END_CASE

	(*FINAL DE FASE 2/3 CASE*)

	(***************************************************************
	Function Block Calls
	***************************************************************)
	(*INICIO DE FASE 3/3 Llamando a funciones*)
	(*Aquí es donde vamos a hacer uso de las funciones. Es la parte que modificaremos y
la que nos va a interesar*)

	(************************** MC_POWER ****************************)
	MC_Power_0.Axis := Axis1Obj;	(* pointer to axis *)
	MC_Power_0();

	(************************** MC_HOME ****************************)
	MC_Home_0.Axis := Axis1Obj;
	MC_Home_0();

	(********************* MC_MOVEABSOLUTE ************************)
	MC_MoveAbsolute_0.Axis := Axis1Obj;
	MC_MoveAbsolute_0();

	(********************* MC_MOVEADDITIVE ************************)
	MC_MoveAdditive_0.Axis := Axis1Obj;
	MC_MoveAdditive_0();

	(********************* MC_MOVEVELOCITY ************************)
	MC_MoveVelocity_0.Axis := Axis1Obj;
	MC_MoveVelocity_0();

	(************************** MC_STOP ****************************)
	MC_Stop_0.Axis := Axis1Obj;
	MC_Stop_0();

	(**************************MC_HALT*****************************)
	MC_Halt_0.Axis := Axis1Obj;
	MC_Halt_0();

	(************************** MC_RESET ****************************)
	MC_Reset_0.Axis := Axis1Obj;
	MC_Reset_0();



	IF NOT DataChangedV THEN

		BasicControl.Parameter.Velocity := NuevaVelocidad;
		BasicControl.Command.MoveVelocity := TRUE;

		DataChangedV := TRUE;

	END_IF





	IF NOT ActionDone THEN

		CASE ReceivedAction OF
			5:
				(************************** MC_POWER
****************************)

				IF MC_Power_0.Status THEN
					BasicControl.Command.Power := FALSE;
```

```
						ELSE
							BasicControl.Command.Power := TRUE;
						END_IF

						ActionDone := TRUE;

					7:
						(************************* MC_HOME
****************************)

						BasicControl.Command.Home := TRUE;
						ActionDone := TRUE;

					9:
						(************************* MC_RESET
***************************)

						BasicControl.Command.ErrorAcknowledge := TRUE;

						ActionDone := TRUE;

					11:

	(**************************MC_HALT***************************)

						BasicControl.Command.Halt := TRUE;

						ActionDone := TRUE;

					12:
						(********************* MC_MOVEVELOCITY
************************)

						BasicControl.Command.MoveVelocity := TRUE;

						ActionDone := TRUE;

					ELSE
						// nothing
				END_CASE

		END_IF
END_PROGRAM
```

2. Módulo Motion2

```
PROGRAM _INIT

(* get axis object *)
Axis2Obj := ADR(gAxis02);

AxisStep := STATE_WAIT; (* start step *)

BasicControl.Parameter.Velocity          := 1000;  (*velocity for movement*)
BasicControl.Parameter.Acceleration      := 5000;  (*acceleration for movement*)
BasicControl.Parameter.Deceleration      := 5000;  (*deceleration for movement*)
BasicControl.Parameter.JogVelocity       := 400;   (*velocity for jogging *)
END_PROGRAM

PROGRAM _CYCLIC

(***************************************************************
```

```
        Control Sequence
****************************************************************)
(* status information is read before the step sequencer to attain a shorter reaction time
*)
(************************ MC_READSTATUS ************************)
MC_ReadStatus_0.Enable := NOT(MC_ReadStatus_0.Error);
MC_ReadStatus_0.Axis := Axis2Obj;
MC_ReadStatus_0();
BasicControl.AxisState.Disabled             := MC_ReadStatus_0.Disabled;
BasicControl.AxisState.StandStill           := MC_ReadStatus_0.StandStill;
BasicControl.AxisState.Stopping             := MC_ReadStatus_0.Stopping;
BasicControl.AxisState.Homing               := MC_ReadStatus_0.Homing;
BasicControl.AxisState.DiscreteMotion       := MC_ReadStatus_0.DiscreteMotion;
BasicControl.AxisState.ContinuousMotion     := MC_ReadStatus_0.ContinuousMotion;
BasicControl.AxisState.SynchronizedMotion   := MC_ReadStatus_0.SynchronizedMotion;
BasicControl.AxisState.ErrorStop            := MC_ReadStatus_0.Errorstop;

(********************MC_BR_READDRIVESTATUS**********************)
MC_BR_ReadDriveStatus_0.Enable := NOT(MC_BR_ReadDriveStatus_0.Error);
MC_BR_ReadDriveStatus_0.Axis := Axis2Obj;
MC_BR_ReadDriveStatus_0.AdrDriveStatus := ADR(BasicControl.Status.DriveStatus);
MC_BR_ReadDriveStatus_0();

(******************** MC_READACTUALPOSITION ********************)
MC_ReadActualPosition_0.Enable := (NOT(MC_ReadActualPosition_0.Error));
MC_ReadActualPosition_0.Axis := Axis2Obj;
MC_ReadActualPosition_0();
IF(MC_ReadActualPosition_0.Valid = TRUE)THEN
    BasicControl.Status.ActPosition := MC_ReadActualPosition_0.Position;
END_IF

(******************** MC_READACTUALVELOCITY ********************)
MC_ReadActualVelocity_0.Enable := (NOT(MC_ReadActualVelocity_0.Error));
MC_ReadActualVelocity_0.Axis := Axis2Obj;
MC_ReadActualVelocity_0();
IF(MC_ReadActualVelocity_0.Valid = TRUE)THEN
    BasicControl.Status.ActVelocity := MC_ReadActualVelocity_0.Velocity;
END_IF

(********************* MC_READACTUALTORQUE *********************)
MC_ReadActualTorque_0.Enable := (NOT(MC_ReadActualTorque_0.Error));
MC_ReadActualTorque_0.Axis :=  Axis2Obj;
MC_ReadActualTorque_0();

IF(MC_ReadActualTorque_0.Valid = TRUE) THEN
               BasicControl.Status.ActTorque := MC_ReadActualTorque_0.Torque;
               ActualTorque := BasicControl.Status.ActTorque;
END_IF

(************************ MC_READAXISERROR *********************)
MC_ReadAxisError_0.Enable := NOT(MC_ReadAxisError_0.Error);
MC_ReadAxisError_0.Axis := Axis2Obj;
MC_ReadAxisError_0.DataAddress := ADR(BasicControl.Status.ErrorText);
MC_ReadAxisError_0.DataLength := SIZEOF(BasicControl.Status.ErrorText);
MC_ReadAxisError_0.DataObjectName := 'acp10etxen';
MC_ReadAxisError_0();

(**************** CHECK FOR GENERAL AXIS ERROR *****************)
    IF ((MC_ReadAxisError_0.AxisErrorID <> 0) AND (MC_ReadAxisError_0.Valid = TRUE)) THEN
        AxisStep := STATE_ERROR_AXIS;
(***************** CHECK IF POWER SHOULD BE OFF ******************)
    ELSIF ((BasicControl.Command.Power = FALSE) AND (MC_ReadAxisError_0.Valid = TRUE)) THEN
        IF ((MC_ReadStatus_0.Errorstop = TRUE) AND (MC_ReadStatus_0.Valid = TRUE)) THEN
            AxisStep := STATE_ERROR_RESET;
        ELSE
            AxisStep := STATE_WAIT;
        END_IF
```

```
    END_IF

(* central monitoring OF stop command attains a shorter reaction TIME in CASE OF emergency
stop *)
(******************CHECK FOR STOP COMMAND************************)
    IF (BasicControl.Command.Stop = TRUE) THEN
        IF ((AxisStep >= STATE_HOME) AND (AxisStep <= STATE_ERROR)) THEN
            (* reset all FB execute inputs we use *)
            MC_Home_0.Execute := 0;
            MC_Stop_0.Execute := 0;
            MC_MoveAbsolute_0.Execute := 0;
            MC_MoveAdditive_0.Execute := 0;
            MC_MoveVelocity_0.Execute := 0;
            MC_ReadAxisError_0.Acknowledge := 0;
            MC_Reset_0.Execute := 0;
            MC_Halt_0.Execute := 0;

            (* reset user commands *)
            BasicControl.Command.Halt := 0;
            BasicControl.Command.Home := 0;
            BasicControl.Command.MoveJogPos := 0;
            BasicControl.Command.MoveJogNeg := 0;
            BasicControl.Command.MoveVelocity := 0;
            BasicControl.Command.MoveAbsolute := 0;
            BasicControl.Command.MoveAdditive := 0;
            AxisStep := STATE_STOP;
        END_IF
    END_IF

CASE AxisStep OF

(******************** WAIT *************************)
    STATE_WAIT:  (* STATE: Wait *)
        IF (BasicControl.Command.Power = TRUE) THEN
            AxisStep := STATE_POWER_ON;
        ELSE
            MC_Power_0.Enable := FALSE;
        END_IF

        (* reset all FB execute inputs we use *)
        MC_Home_0.Execute := FALSE;
        MC_Stop_0.Execute := FALSE;
        MC_MoveAbsolute_0.Execute := FALSE;
        MC_MoveAdditive_0.Execute := FALSE;
        MC_MoveVelocity_0.Execute := FALSE;
        MC_Halt_0.Execute := FALSE;
        MC_ReadAxisError_0.Acknowledge := FALSE;
        MC_Reset_0.Execute := FALSE;

        (* reset user commands *)
        BasicControl.Command.Stop := FALSE;
        BasicControl.Command.Halt := FALSE;
        BasicControl.Command.Home := FALSE;
        BasicControl.Command.MoveJogPos := FALSE;
        BasicControl.Command.MoveJogNeg := FALSE;
        BasicControl.Command.MoveVelocity := FALSE;
        BasicControl.Command.MoveAbsolute := FALSE;
        BasicControl.Command.MoveAdditive := FALSE;

        BasicControl.Status.ErrorID := 0;

(******************** POWER ON *********************)
    STATE_POWER_ON:  (* STATE: Power on *)
        MC_Power_0.Enable := TRUE;
        IF (MC_Power_0.Status = TRUE) THEN
            AxisStep := STATE_READY;
        END_IF
```

```
        (* if a power error occured go to error state *)
        IF (MC_Power_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_Power_0.ErrorID;
            AxisStep := STATE_ERROR;
        END_IF

(******************* READY *********************)
    STATE_READY:  (* STATE: Waiting for commands *)
        IF (BasicControl.Command.Home = TRUE)THEN
            BasicControl.Command.Home := FALSE;
            AxisStep := STATE_HOME;

        ELSIF (BasicControl.Command.Stop = TRUE) THEN
            AxisStep := STATE_STOP;

        ELSIF (BasicControl.Command.MoveJogPos = TRUE) THEN
            AxisStep := STATE_JOG_POSITIVE;

        ELSIF (BasicControl.Command.MoveJogNeg = TRUE) THEN
            AxisStep := STATE_JOG_NEGATIVE;

        ELSIF (BasicControl.Command.MoveAbsolute = TRUE) THEN
            BasicControl.Command.MoveAbsolute := FALSE;
            AxisStep := STATE_MOVE_ABSOLUTE;

        ELSIF (BasicControl.Command.MoveAdditive = TRUE) THEN
            BasicControl.Command.MoveAdditive := FALSE;
            AxisStep := STATE_MOVE_ADDITIVE;

        ELSIF (BasicControl.Command.MoveVelocity = TRUE) THEN
            BasicControl.Command.MoveVelocity := FALSE;
            AxisStep := STATE_MOVE_VELOCITY;

        ELSIF (BasicControl.Command.Halt = TRUE) THEN
            BasicControl.Command.Halt := FALSE;
                      AxisStep := STATE_HALT;

               ELSIF BasicControl.Command.Torque THEN
                      BasicControl.Command.Torque := FALSE;
                      AxisStep := STATE_TORQUE;
               END_IF

(******************* HOME *********************)
    STATE_HOME:  (* STATE: start homing process *)
        MC_Home_0.Position := BasicControl.Parameter.HomePosition;
        MC_Home_0.HomingMode := BasicControl.Parameter.HomeMode;
        MC_Home_0.Execute := TRUE;
        IF (MC_Home_0.Done = TRUE) THEN
            MC_Home_0.Execute := FALSE;
            AxisStep := STATE_READY;
        END_IF
        (* if a homing error occured go to error state *)
        IF (MC_Home_0.Error = TRUE) THEN
            MC_Home_0.Execute := FALSE;
            BasicControl.Status.ErrorID := MC_Home_0.ErrorID;
            AxisStep := STATE_ERROR;
        END_IF

(**********************HALT_MOVEMENT*************************)
    STATE_HALT:   (* STATE: Halt movement *)
        MC_Halt_0.Deceleration := BasicControl.Parameter.Deceleration;
        MC_Halt_0.Execute := TRUE;
        IF (MC_Halt_0.Done = TRUE) THEN
            MC_Halt_0.Execute := FALSE;
            AxisStep := STATE_READY;
        END_IF
        (* check if error occured *)
```

```
        IF (MC_Halt_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_Halt_0.ErrorID;
            MC_Halt_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(*********************** STOP MOVEMENT ************************)
    STATE_STOP:  (* STATE: Stop movement *)
        MC_Stop_0.Deceleration := BasicControl.Parameter.Deceleration;
        MC_Stop_0.Execute := TRUE;
        (* if axis is stopped go to ready state *)
        IF ((MC_Stop_0.Done = TRUE) AND (BasicControl.Command.Stop = FALSE)) THEN
            MC_Stop_0.Execute := FALSE;
            AxisStep := STATE_READY;
        END_IF
        (* check if error occured *)
        IF (MC_Stop_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_Stop_0.ErrorID;
            MC_Stop_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(******************** START JOG MOVEMENT POSITVE *********************)
     STATE_JOG_POSITIVE:  (* STATE: Start jog movement in positive direction *)
        MC_MoveVelocity_0.Velocity      := BasicControl.Parameter.JogVelocity;
        MC_MoveVelocity_0.Acceleration  := BasicControl.Parameter.Acceleration;
        MC_MoveVelocity_0.Deceleration  := BasicControl.Parameter.Deceleration;
        MC_MoveVelocity_0.Direction     := mcPOSITIVE_DIR;
        MC_MoveVelocity_0.Execute := TRUE;
        IF (BasicControl.Command.MoveJogPos = FALSE) THEN
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_HALT;
        END_IF
        (* check if error occured *)
        IF (MC_MoveVelocity_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_MoveVelocity_0.ErrorID;
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(******************** START JOG MOVEMENT NEGATIVE *********************)
    STATE_JOG_NEGATIVE:  (* STATE: Start jog movement in negative direction *)
        MC_MoveVelocity_0.Velocity      := BasicControl.Parameter.JogVelocity;
        MC_MoveVelocity_0.Acceleration  := BasicControl.Parameter.Acceleration;
        MC_MoveVelocity_0.Deceleration  := BasicControl.Parameter.Deceleration;
        MC_MoveVelocity_0.Direction      := mcNEGATIVE_DIR;
        MC_MoveVelocity_0.Execute := TRUE;
        IF (BasicControl.Command.MoveJogNeg = FALSE) THEN
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_HALT;
        END_IF
        (* check if error occured *)
        IF (MC_MoveVelocity_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_MoveVelocity_0.ErrorID;
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(******************** START ABSOLUTE MOVEMENT *********************)
     STATE_MOVE_ABSOLUTE:  (* STATE: Start absolute movement *)
        MC_MoveAbsolute_0.Position      := BasicControl.Parameter.Position;
        MC_MoveAbsolute_0.Velocity      := BasicControl.Parameter.Velocity;
        MC_MoveAbsolute_0.Acceleration  := BasicControl.Parameter.Acceleration;
        MC_MoveAbsolute_0.Deceleration  := BasicControl.Parameter.Deceleration;
        MC_MoveAbsolute_0.Direction     := BasicControl.Parameter.Direction;
        MC_MoveAbsolute_0.Execute := TRUE;
        (* check if commanded position is reached *)
```

```
        IF (BasicControl.Command.Halt) THEN
            BasicControl.Command.Halt := FALSE;
            MC_MoveAbsolute_0.Execute := FALSE;
            AxisStep := STATE_HALT;
        ELSIF (MC_MoveAbsolute_0.Done = TRUE) THEN
            MC_MoveAbsolute_0.Execute := FALSE;
            AxisStep := STATE_READY;
        END_IF
        (* check if error occured *)
        IF (MC_MoveAbsolute_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_MoveAbsolute_0.ErrorID;
            MC_MoveAbsolute_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(******************* START ADDITIVE MOVEMENT *********************)
    STATE_MOVE_ADDITIVE:  (* STATE: Start additive movement *)
        MC_MoveAdditive_0.Distance      := BasicControl.Parameter.Distance;
        MC_MoveAdditive_0.Velocity      := BasicControl.Parameter.Velocity;
        MC_MoveAdditive_0.Acceleration  := BasicControl.Parameter.Acceleration;
        MC_MoveAdditive_0.Deceleration  := BasicControl.Parameter.Deceleration;
        MC_MoveAdditive_0.Execute := TRUE;
        (* check if commanded distance is reached *)
        IF (BasicControl.Command.Halt) THEN
            BasicControl.Command.Halt := FALSE;
            MC_MoveAdditive_0.Execute := FALSE;
            AxisStep := STATE_HALT;
        ELSIF (MC_MoveAdditive_0.Done = TRUE) THEN
            MC_MoveAdditive_0.Execute := FALSE;
            AxisStep := STATE_READY;
        END_IF
        (* check if error occured *)
        IF (MC_MoveAdditive_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_MoveAdditive_0.ErrorID;
            MC_MoveAdditive_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(******************* START VELOCITY MOVEMENT *********************)
    STATE_MOVE_VELOCITY:  (* STATE: Start velocity movement *)
        MC_MoveVelocity_0.Velocity      := BasicControl.Parameter.Velocity;
        MC_MoveVelocity_0.Acceleration  := BasicControl.Parameter.Acceleration;
        MC_MoveVelocity_0.Deceleration  := BasicControl.Parameter.Deceleration;
        MC_MoveVelocity_0.Direction     := BasicControl.Parameter.Direction;
        MC_MoveVelocity_0.Execute := TRUE;
        (* check if commanded velocity is reached *)
        IF (BasicControl.Command.Halt) THEN
            BasicControl.Command.Halt := FALSE;
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_HALT;
        ELSIF (MC_MoveVelocity_0.InVelocity = TRUE) THEN
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_READY;
        END_IF
        (* check if error occured *)
        IF (MC_MoveVelocity_0.Error = TRUE) THEN
            BasicControl.Status.ErrorID := MC_MoveVelocity_0.ErrorID;
            MC_MoveVelocity_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF

(******************* APPLY TORQUE *********************)
       STATE_TORQUE:

              // Resto de parámetros por ver
              MC_TorqueControl_0.Execute := TRUE;
              MC_TorqueControl_0.Torque := BasicControl.Parameter.Torque
```

```
            MC_TorqueControl_0.Acceleration:= 0;
            MC_TorqueControl_0.TorqueRamp := 0.5;
            MC_TorqueControl_0.Velocity := 20000;

            // Viendo si el par requerido se ha logrado ya.
            IF MC_TorqueControl_0.InTorque THEN
                   MC_MoveVelocity_0.Execute := FALSE;
                   AxisStep := STATE_READY;
            END_IF
            // Confirmar si se produce error
            IF MC_TorqueControl_0.Error THEN
                   BasicControl.Status.ErrorID := MC_TorqueControl_0.ErrorID;
                   MC_TorqueControl_0.Execute := FALSE;
                   AxisStep := STATE_ERROR;
            END_IF


(******************** FB-ERROR OCCURED ************************)
    STATE_ERROR:  (* STATE: Error *)
        (* check if FB indicates an axis error *)
        IF (MC_ReadAxisError_0.AxisErrorCount<>0) THEN
            AxisStep := STATE_ERROR_AXIS;
        ELSE
            IF (BasicControl.Command.ErrorAcknowledge = TRUE) THEN
                BasicControl.Command.ErrorAcknowledge := FALSE;
                BasicControl.Status.ErrorID := 0;
                (* reset axis if it is in axis state ErrorStop *)
                IF ((MC_ReadStatus_0.Errorstop = TRUE) AND (MC_ReadStatus_0.Valid = TRUE))
THEN
                    AxisStep := STATE_ERROR_RESET;
                ELSE
                    AxisStep := STATE_WAIT;
                END_IF
            END_IF
        END_IF

(******************** AXIS-ERROR OCCURED ************************)
    STATE_ERROR_AXIS:  (* STATE: Axis Error *)
        IF (MC_ReadAxisError_0.Valid = TRUE) THEN
            IF (MC_ReadAxisError_0.AxisErrorID <> 0) THEN
                BasicControl.Status.ErrorID := MC_ReadAxisError_0.AxisErrorID;
            END_IF
            MC_ReadAxisError_0.Acknowledge := FALSE;
            IF (BasicControl.Command.ErrorAcknowledge = TRUE) THEN
                BasicControl.Command.ErrorAcknowledge := FALSE;
                (* acknowledge axis error *)
                IF (MC_ReadAxisError_0.AxisErrorID <> 0) THEN
                    MC_ReadAxisError_0.Acknowledge := TRUE;
                END_IF
            END_IF
            IF (MC_ReadAxisError_0.AxisErrorCount = 0) THEN
                (* reset axis if it is in axis state ErrorStop *)
                BasicControl.Status.ErrorID := 0;
                IF ((MC_ReadStatus_0.Errorstop = TRUE) AND (MC_ReadStatus_0.Valid = TRUE))
THEN
                    AxisStep := STATE_ERROR_RESET;
                ELSE
                    AxisStep := STATE_WAIT;
                END_IF
            END_IF
        END_IF

(******************** RESET DONE ************************)
    STATE_ERROR_RESET:  (* STATE: Wait for reset done *)
        MC_Reset_0.Execute := TRUE;
        (* reset MC_Power.Enable if this FB is in Error*)
        IF (MC_Power_0.Error = TRUE) THEN
```

```
            MC_Power_0.Enable := FALSE;
        END_IF
        IF(MC_Reset_0.Done = TRUE)THEN
            MC_Reset_0.Execute := FALSE;
            AxisStep := STATE_WAIT;
        ELSIF(MC_Reset_0.Error = TRUE) THEN
            MC_Reset_0.Execute := FALSE;
            AxisStep := STATE_ERROR;
        END_IF
(******************** SEQUENCE END *************************)
END_CASE

(****************************************************************
        Function Block Calls
****************************************************************)

(************************** MC_POWER ****************************)
       MC_Power_0.Axis := Axis2Obj;  (* pointer to axis *)
       MC_Power_0();

       (************************** MC_HOME *****************************)
       MC_Home_0.Axis := Axis2Obj;
       MC_Home_0();

       (********************** MC_MOVEABSOLUTE *************************)
       MC_MoveAbsolute_0.Axis := Axis2Obj;
       MC_MoveAbsolute_0();

       (********************** MC_MOVEADDITIVE *************************)
       MC_MoveAdditive_0.Axis := Axis2Obj;
       MC_MoveAdditive_0();

       (********************** MC_MOVEVELOCITY *************************)
       MC_MoveVelocity_0.Axis := Axis2Obj;
       MC_MoveVelocity_0();

       (************************** MC_STOP *****************************)
       MC_Stop_0.Axis := Axis2Obj;
       MC_Stop_0();

       (**************************MC_HALT******************************)
       MC_Halt_0.Axis := Axis2Obj;
       MC_Halt_0();

       (************************** MC_RESET ****************************)
       MC_Reset_0.Axis := Axis2Obj;
       MC_Reset_0();

       (************************** MC_TORQUECONTROL ****************************)
       MC_TorqueControl_0.Axis := Axis2Obj;
       MC_TorqueControl_0();

       IF NOT DataChangedT THEN

              BasicControl.Parameter.Torque := NuevoPar;
              BasicControl.Command.Torque := TRUE;
              DataChangedT := 1;

       END_IF

       IF NOT ActionDone THEN

              CASE ReceivedAction OF
                     6:
                            IF MC_Power_0.Status THEN
                                   BasicControl.Command.Power := TRUE;
                            ELSE
```

```
                                BasicControl.Command.Power := FALSE;
                        END_IF

                        ActionDone := TRUE;

                8:
                        BasicControl.Command.Power := TRUE;

                        ActionDone := TRUE;

                10:
                        BasicControl.Command.ErrorAcknowledge := 0;

                        ActionDone := TRUE;

                ELSE
                        // nothing
        END_CASE

    END_IF
END_PROGRAM
```

3. Módulo Server

```
PROGRAM _INIT

    (* Server configuration *)
    gServer.ParaPort := 12010; (* TCP port on the server *)
    gServer.ParaSendTime := T#100ms;
    gServer.ParaReceiveTimeout := T#750ms;

    (* Client communication timing configuration *)
    (* Read cycle time *)
    GetRTInfo(enable := 1);
    RTResult.Status := GetRTInfo.status;
    RTResult.CycleTime := GetRTInfo.cycle_time;
    RTResult.TaskClass := GetRTInfo.task_class;
    RTResult.InitReason := GetRTInfo.init_reason;

    (* Start Logger *)
    (* Move all entries *)
    FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO
        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
    END_FOR;
    (* Empty Log[0] *)
    brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
    (* Write on Log[0] *)
    GetTimeStruct(enable := 1,pDTStructure := ADR(ServerLog[0].Date));
    ServerLog[0].FBK := 'NA';
    ServerLog[0].ErrorID := 0;
    gServer.Enable := TRUE; // Importante para que al entrar en nuestro ciclo entre ya
con el server abierto.

END_PROGRAM


PROGRAM _CYCLIC

    (* Send and timeout timer calculation *)
    IF (TIME_TO_UDINT(gServer.ParaSendTime) < (RTResult.CycleTime / 1000) *
SEND_TIME_MULT) AND RTResult.CycleTime > 0 THEN
        gServer.ParaSendTime := (RTResult.CycleTime / 1000) * SEND_TIME_MULT;
    END_IF;
```

```
        IF (TIME_TO_UDINT(gServer.ParaReceiveTimeout) < (RTResult.CycleTime / 1000) *
TIMEOUT_MULT) AND RTResult.CycleTime > 0 THEN
                gServer.ParaReceiveTimeout := (RTResult.CycleTime / 1000) * TIMEOUT_MULT;
        END_IF;


        CASE Server.sStep OF

                0:
                        IF gServer.Enable THEN // En nuestro caso viene ya cargado de la
inicialización
                                gServer.Status := 1; // WAITING FOR FIRST COMMUNICATION
                                Server.sStep := 5;
                        END_IF;

                5:  (* Open Ethernet Interface *) // Escucha en el puerto que hemos dicho
que iba a mandar info
                        Server.TcpOpen_0.enable := 1;
                        Server.TcpOpen_0.pIfAddr := 0;  (* Listen on all TCP/IP Interfaces*)
                        Server.TcpOpen_0.port := gServer.ParaPort;  (* Port to listen*)
                        Server.TcpOpen_0.options := tcpOPT_REUSEADDR;      (* Allows the
linking of several instances of a TCP server with the same port *)
                        Server.TcpOpen_0;  (* Call the Function*)

                        IF Server.TcpOpen_0.status = 0 THEN  (* TcpOpen successfull*)
                                Server.sStep := 6;
                        ELSIF Server.TcpOpen_0.status = tcpERR_ALREADY_EXIST OR
gServer.Enable = 0 THEN
                                (* Log error *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpOpen_0';
                                ServerLog[0].ErrorID := Server.TcpOpen_0.status;

                                (* Close communication *)
                                Server.sStep := 50;
                        ELSIF Server.TcpOpen_0.status = ERR_FUB_BUSY THEN  (* TcpOpen not
finished -> redo *)
                                (* Busy *)
                        ELSE
                                (* Log error *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpOpen_0';
                                ServerLog[0].ErrorID := Server.TcpOpen_0.status;

                        END_IF

                6:
                        Server.linger_opt.lLinger := 0; (* linger Time = 0 *)
                        Server.linger_opt.lOnOff := 1; (* linger Option ON *)

                        Server.TcpIoctl_0.enable := 1;
                        Server.TcpIoctl_0.ident := Server.TcpOpen_0.ident; (* Connection
```

```
Ident from AsTCP.TCP_Open *)
                      Server.TcpIoctl_0.ioctl := tcpSO_LINGER_SET; (* Set Linger Options *)
                      Server.TcpIoctl_0.pData := ADR(Server.linger_opt);
                      Server.TcpIoctl_0.datalen := SIZEOF(Server.linger_opt);
                      Server.TcpIoctl_0;

                      IF Server.TcpIoctl_0.status = 0 THEN  (* TcpIoctl successfull *)
                             Server.sStep := 10;

                      ELSIF Server.TcpIoctl_0.status = ERR_FUB_BUSY THEN  (* TcpIoctl not
finished -> redo *)
                             (* Busy *)
                      ELSE
                             (* Log error *)
                             FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

       brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                             END_FOR;
                             brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                             GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                             ServerLog[0].FBK := 'TcpIoctl_0';
                             ServerLog[0].ErrorID := Server.TcpIoctl_0.status;
                      END_IF

               10: (* Wait for Client Connection *)
                      Server.TcpServer_0.enable := 1;
                      Server.TcpServer_0.ident := Server.TcpOpen_0.ident;  (* Connection
Ident from AsTCP.TCP_Open *)
                      Server.TcpServer_0.backlog := 1;  (* Number of clients waiting
simultaneously for a connection*)
                      Server.TcpServer_0.pIpAddr := ADR(Server.client_address);  (* Where
to write the client IP-Address*)
                      Server.TcpServer_0;  (* Call the Function*)

                      IF Server.TcpServer_0.status = 0 THEN  (* Status = 0 if an client
connects to server *)
                             Server.sStep := 15;
                      ELSIF Server.TcpServer_0.status = tcpERR_INVALID_IDENT OR
gServer.Enable = 0 THEN
                             (* Log error *)
                             FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

       brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                             END_FOR;
                             brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                             GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                             ServerLog[0].FBK := 'TcpServer_0';
                             ServerLog[0].ErrorID := Server.TcpServer_0.status;

                             (* Close communication *)
                             Server.sStep := 50;
                      ELSIF Server.TcpServer_0.status = ERR_FUB_BUSY THEN  (* TcpServer not
finished -> redo *)
                             (* Busy *)
                      ELSE
                             (* Log error *)
                             FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

       brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                             END_FOR;
                             brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                             GetTimeStruct(enable := 1,pDTStructure :=
```

```
ADR(ServerLog[0].Date));
                              ServerLog[0].FBK := 'TcpServer_0';
                              ServerLog[0].ErrorID := Server.TcpServer_0.status;
                      END_IF

              15:
                      Server.TcpIoctl_0.enable := 1;
                      Server.TcpIoctl_0.ident := Server.TcpServer_0.identclnt; (*
Connection Ident from AsTCP.TCP_Server *)
                      Server.TcpIoctl_0.ioctl := tcpSO_LINGER_SET;  (* Set Linger Options
*)
                      Server.TcpIoctl_0.pData := ADR(Server.linger_opt);
                      Server.TcpIoctl_0.datalen := SIZEOF(Server.linger_opt);
                      Server.TcpIoctl_0;

                      IF Server.TcpIoctl_0.status = 0 THEN  (* TcpIoctl successfull *) //
Aquí ya todo esta listo para el envío y la recepción de datos
                              (* Start send timer *)
                              TON_Send_Server.IN := 1;
                              TON_Send_Server.PT := gServer.ParaSendTime;

                              Server.sStep := 20;

                      ELSIF Server.TcpIoctl_0.status = ERR_FUB_BUSY THEN  (* TcpIoctl not
finished -> redo *)
                              (* Busy *)
                      ELSE
                              (* Log error *)
                              FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

       brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                              END_FOR;
                              brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                              GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                              ServerLog[0].FBK := 'TcpIoctl_0';
                              ServerLog[0].ErrorID := Server.TcpIoctl_0.status;
                      END_IF

              20: (* Wait for Data *)
                      Server.TcpRecv_0.enable := 1;
                      Server.TcpRecv_0.ident := Server.TcpServer_0.identclnt;  (* Client
Ident from AsTCP.TCP_Server *)
                      Server.TcpRecv_0.pData:= ADR(Server.data_buffer);  (* Where to store
the incoming data *)
                      Server.TcpRecv_0.datamax := SIZEOF(Server.data_buffer);  (* Length of
data buffer *)
                      Server.TcpRecv_0.flags := 0;
                      Server.TcpRecv_0;  (* Call the Function*)

                      TON_RecvTimeout_Server.IN := 1;
                      TON_RecvTimeout_Server.PT := gServer.ParaReceiveTimeout;

                      IF gServer.Enable = 0 OR (Server.TcpRecv_0.status =
tcpERR_NOT_CONNECTED) THEN
                              (* Log error *)
                              FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

       brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                              END_FOR;
                              brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                              GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                              ServerLog[0].FBK := 'TcpRecv_0';
                              ServerLog[0].ErrorID := Server.TcpRecv_0.status;
```

```
                                (* Connection lost or disabled *)
                                Server.sStep := 40;
                        ELSIF Server.TcpRecv_0.status = 0 THEN  (* Data received *)
                                gServer.Status := 0; //COMMUNICATION OK
                                (* Reset timeout timer, since data was received *)
                                // Si ha recibido data, la almacemos y marcamos que se ha
recibido

                                ResetValue := UDINT_TO_UINT(Server.data_buffer[0]);
                                received := TRUE;

                                IF ResetValue.15 THEN

                                        ReceivedData := (ResetValue AND 16#EF);
                                        DataChanged := 0;

                                ELSIF NOT ResetValue.15 AND ActionDone THEN

                                        ReceivedAction := ResetValue;
                                        ActionDone :=FALSE;

                                END_IF



                                TON_RecvTimeout_Server.IN := 0;

                                (* Time for new communication *)
                                IF TON_Send_Server.Q THEN
                                        TON_Send_Server.IN := 0;
                                        (* Go to send step *) // Si no se ha recibido nada en
un tiempo limite pasamos a leer.
                                        Server.sStep := 30;
                                END_IF;
                        ELSIF Server.TcpRecv_0.status = tcpERR_NO_DATA THEN
                                (* No data received - wait for data, timeout or new
communication.
                                   The client will send data again, while the RecvTimeout
timer is
                                   still counting. If this timeout is reached, an entry will
be
                                   written in the communication logger. *)

                                IF TON_Send_Server.Q THEN
                                        (* Reset communication timer *)
                                        TON_Send_Server.IN := 0;

                                        (* Go to send step *)
                                        Server.sStep := 30;
                                END_IF;

                                IF TON_RecvTimeout_Server.Q THEN
                                        gServer.Status := 2; //COMMUNICATION ERROR: TIMEOUT
                                        (* Log error *)
                                        FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                        END_FOR;
                                        brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                        GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                        ServerLog[0].FBK := 'TcpRecv_0';
                                        ServerLog[0].ErrorID := Server.TcpRecv_0.status;

                                        (* Reset timeout counter and keep waiting for
```

```
communication timer *)
                                        TON_RecvTimeout_Server.IN := 0;

                                        (* Stay in the TcpRecv step until a new communication
is demanded *)
                                END_IF;
                        ELSIF Server.TcpRecv_0.status = ERR_FUB_BUSY THEN  (* TcpRecv not
finished -> redo *)
                                (* Busy *)
                        ELSE
                                (* Log error and wait for new send command *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpRecv_0';
                                ServerLog[0].ErrorID := Server.TcpRecv_0.status;

                                (* Time for new communication *)
                                IF TON_Send_Server.Q THEN
                                        TON_Send_Server.IN := 0;
                                        (* Go to send step *)
                                        Server.sStep := 30;
                                END_IF;
                        END_IF



                30:     (* Send Data back to Client *)       // UaCounter será nuestra manera
de introducir data
                        Server.TcpSend_0.enable := 1;
                        Server.TcpSend_0.ident := Server.TcpServer_0.identclnt;  (* Client
Ident from AsTCP.TCP_Server *)

                        IF alterner THEN
                                SentData := TrueVelocity;
                                (*Alterner := FALSE;
                        ELSE
                                SentData := TrueTorque;
                                Alterner := TRUE;*)
                        END_IF

                        Server.TcpSend_0.pData := ADR(SentData);  (* Which data to send *)
                        Server.TcpSend_0.datalen := SIZEOF(SentData);  (* Lenght of data to
send *)
                        Server.TcpSend_0.flags := 0;
                        Server.TcpSend_0;  (* Call the Function*)

                        IF gServer.Enable = 0 OR (Server.TcpSend_0.status =
tcpERR_NOT_CONNECTED) THEN
                                (* Log error *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpSend_0';
                                ServerLog[0].ErrorID := Server.TcpSend_0.status;

                                (* Connection lost or disabled *)
```

```
						Server.sStep := 40;
					ELSIF Server.TcpSend_0.status = 0 THEN  (* Data sent *)
						TON_Send_Server.IN := 1;
						TON_Send_Server.PT := gServer.ParaSendTime;
						Server.sStep := 20;
						IF TON_Send_Server.Q THEN
							TON_Send_Server.IN := 0;
							(* Go to send step *)
							Server.sStep := 30;
						END_IF;

					ELSIF Server.TcpSend_0.status = ERR_FUB_BUSY OR
Server.TcpSend_0.status = tcpERR_WOULDBLOCK THEN  (* TcpSend not finished -> redo *)

						(* Busy *)
					ELSIF Server.TcpSend_0.status = tcpERR_SENTLEN THEN
						(* Log error *)
						FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

	brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
						END_FOR;
						brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
						GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
						ServerLog[0].FBK := 'TcpSend_0';
						ServerLog[0].ErrorID := Server.TcpSend_0.status;

						(* Calculate leftover data *)
						pLeftoverData := Server.TcpSend_0.pData +
Server.TcpSend_0.sentlen;
						LeftoverDataSize := Server.TcpSend_0.datalen -
Server.TcpSend_0.sentlen;

						(* Go to a new Send state, where leftover data will be sent *)
						Server.sStep := 35;
					ELSE
						(* Log error *)
						FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

	brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
						END_FOR;
						brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
						GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
						ServerLog[0].FBK := 'TcpSend_0';
						ServerLog[0].ErrorID := Server.TcpSend_0.status;
					END_IF

				35:
					(* Sent leftover data *)
					Server.TcpSend_0.enable := 1;
					Server.TcpSend_0.ident := Server.TcpServer_0.identclnt;  (* Client
Ident from AsTCP.TCP_Server *)
					Server.TcpSend_0.pData := pLeftoverData;  (* Which data to send *)
					Server.TcpSend_0.datalen := LeftoverDataSize;  (* Lenght of data to
send *)
					Server.TcpSend_0.flags := 0;
					Server.TcpSend_0;  (* Call the Function*)

					IF gServer.Enable = 0 OR (Server.TcpSend_0.status =
tcpERR_NOT_CONNECTED) THEN
						(* Log error *)
						FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

	brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
```

```
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpSend_0';
                                ServerLog[0].ErrorID := Server.TcpSend_0.status;

                                (* Connection lost or disabled *)
                                Server.sStep := 40;
                        ELSIF Server.TcpSend_0.status = 0 THEN  (* Data sent *)
                                TON_Send_Server.IN := 1;
                                TON_Send_Server.PT := gServer.ParaSendTime;
                                Server.sStep := 20;
                        ELSIF Server.TcpSend_0.status = ERR_FUB_BUSY OR
Server.TcpSend_0.status = tcpERR_WOULDBLOCK THEN  (* TcpSend not finished -> redo *)

                                (* Busy *)
                        ELSIF Server.TcpSend_0.status = tcpERR_SENTLEN THEN
                                (* Log error *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpSend_0';
                                ServerLog[0].ErrorID := Server.TcpSend_0.status;

                                (* Calculate leftover data *)
                                pLeftoverData := Server.TcpSend_0.pData +
Server.TcpSend_0.sentlen;
                                LeftoverDataSize := Server.TcpSend_0.datalen -
Server.TcpSend_0.sentlen;

                                (* Go to a new Send state, where leftover data will be sent *)
                                Server.sStep := 35;
                        ELSE
                                (* Log error *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

        brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                                END_FOR;
                                brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                                GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                                ServerLog[0].FBK := 'TcpSend_0';
                                ServerLog[0].ErrorID := Server.TcpSend_0.status;
                        END_IF

                40:
                        Server.TcpClose_0.enable := 1;
                        Server.TcpClose_0.ident := Server.TcpServer_0.identclnt;
                        Server.TcpClose_0.how := 0;; // tcpSHUT_RD OR tcpSHUT_WR;
                        Server.TcpClose_0;

                        IF Server.TcpClose_0.status = 0 THEN
                                Server.sStep := 50;
                        ELSIF Server.TcpClose_0.status = ERR_FUB_BUSY THEN  (* TcpClose not
finished -> redo *)
                                (* Busy *)
                        ELSIF Server.TcpClose_0.status = tcpERR_INVALID_IDENT THEN;
                                (* Log error *)
                                FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO
```

```
                              brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                              END_FOR;
                              brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                              GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                              ServerLog[0].FBK := 'TcpClose_0';
                              ServerLog[0].ErrorID := Server.TcpClose_0.status;

                              (* Close server port *)
                              Server.sStep := 50;
                       END_IF
               50:
                       Server.TcpClose_0.enable := 1;
                       Server.TcpClose_0.ident := Server.TcpOpen_0.ident;
                       Server.TcpClose_0.how := 0; //tcpSHUT_RD OR tcpSHUT_WR;
                       Server.TcpClose_0;

                       IF Server.TcpClose_0.status = 0 THEN
                              Server.sStep := 0;
                       ELSIF Server.TcpClose_0.status = ERR_FUB_BUSY THEN  (* TcpClose not
finished -> redo *)
                              (* Busy *)
                       ELSE
                              (* Log error *)
                              FOR iLogEntry := MAX_ENTRIES TO 1 BY -1 DO

       brsmemcpy(ADR(ServerLog[iLogEntry]),ADR(ServerLog[iLogEntry-
1]),SIZEOF(ServerLog[iLogEntry]));
                              END_FOR;
                              brsmemset(ADR(ServerLog[0]),0,SIZEOF(ServerLog[0]));
                              GetTimeStruct(enable := 1,pDTStructure :=
ADR(ServerLog[0].Date));
                              ServerLog[0].FBK := 'TcpClose_0';
                              ServerLog[0].ErrorID := Server.TcpClose_0.status;
                       END_IF

       END_CASE

       TON_Send_Server;
       TON_RecvTimeout_Server;

END_PROGRAM

PROGRAM _EXIT

       REPEAT
               Server.TcpClose_0.enable := 1;
               Server.TcpClose_0.ident := Server.TcpServer_0.identclnt;
               Server.TcpClose_0.how := 0; //tcpSHUT_RD OR tcpSHUT_WR;
               Server.TcpClose_0;

               UNTIL
               Server.TcpClose_0.status <> ERR_FUB_BUSY
       END_REPEAT;

       REPEAT
               Server.TcpClose_0.enable := 1;
               Server.TcpClose_0.ident := Server.TcpOpen_0.ident;
               Server.TcpClose_0.how := 0; //tcpSHUT_RD OR tcpSHUT_WR;
               Server.TcpClose_0;

               UNTIL
               Server.TcpClose_0.status <> ERR_FUB_BUSY
       END_REPEAT;
```

```
END_PROGRAM
```

4. Módulo Program

```
PROGRAM _INIT
        (* Iniicalización de variables*)
        Enable1:=0;
        Enable2:=0;
        DataChangedV := 0;
        DataChangedT := 0;
        NuevoPar := 0;
        NuevaVelocidad := 0;
        LReceivedData := 0;
END_PROGRAM

PROGRAM _CYCLIC

        // Pasamos variables globales a locales para trabajar con ellas.

        LReceivedData := ReceivedData;

        //Actualización de datos de par y velocidad en caso de ser cambiados

        IF NOT DataChanged THEN

                DataChanged := 1;

                IF LReceivedData.14 THEN

                        NuevoPar := PAR_MAX*(LReceivedData AND 16#DF)/100;
                        DataChangedT := 0;

                ELSIF NOT LReceivedData.14 THEN

                        NuevaVelocidad := VEL_MAX*(LReceivedData AND 16#EF)/100;
                        DataChangedV := 0;

                END_IF

        END_IF



        // ENABLE - Habilitación de motores por código
        IF ReceivedAction = 1 THEN

                Enable1:=TRUE;

        ELSIF ReceivedAction = 2 THEN

                Enable2:=TRUE;

        END_IF



        // DISABLE - Deshabilitacion de motores por codigo
        IF ReceivedAction = 3 THEN

                Enable1:=FALSE;
```

```
        ELSIF ReceivedAction = 4 THEN

               Enable2:=FALSE;

        END_IF


        // Manage data sent.

        TrueVelocity := TRUNC(ActualVelocity*VEL_CONV);

        TrueTorque := TRUNC(ActualTorque*TORQ_CONV) OR 16#80;


END_PROGRAM

PROGRAM _EXIT
        (* Insert code here *)

END_PROGRAM
```