Universidad Católica de Santa María

Facultad de Ciencias e Ingenierías Físicas y Formales

Escuela Profesional de Ingeniería Mecánica, Mecánica-Eléctrica y Mecatrónica

DISEÑO E IMPLEMENTACIÓN DE UN SOFTWARE COMPATIBLE CON ARDUINO BASADO EN EL ESTÁNDAR IEC 61131-3 PARA PROGRAMACIÓN LADDER Y SUPERVISIÓN MEDIANTE UN SISTEMA SCADA.

Tesis presentada por el Bachiller:

Medina Zeballos, Diego Alonso

Para optar el Título Profesional de:

Ingeniero Mecatrónico

Asesor:

Mg. Mestas Ramos, Sergio Orlando

Arequipa - Perú

2022

DICTAMEN APROBATORIO

UCSM-ERP

UNIVERSIDAD CATÓLICA DE SANTA MARÍA

INGENIERIA MECANICA, MECANICA-ELECTRICA Y MECATRONICA

TITULACIÓN CON TESIS

DICTAMEN APROBACIÓN DE BORRADOR

Arequipa, 25 de Abril del 2022

Dictamen: 001566-C-EPIMMEM-2022

Visto el borrador del expediente 001566, presentado por:

2013601231 - MEDINA ZEBALLOS DIEGO ALONSO

Titulado:

DISEÑO E IMPLEMENTACIÓN DE UN SOFTWARE COMPATIBLE CON ARDUINO BASADO EN EL ESTÁNDAR IEC 61131-3 PARA PROGRAMACIÓN LADDER Y SUPERVISIÓN MEDIANTE UN SISTEMA SCADA

Nuestro dictamen es:

APROBADO

2213 - QUISPE CCACHUCO MARCELO JAIME
DICTAMINADOR

2397 - CUADROS MACHUCA JUAN CARLOS
DICTAMINADOR

2776 - COLLADO OPORTO CHRISTIAM GUILLERMO
DICTAMINADOR

DEDICATORIA

A mis padres Lorenzo e Yris, quienes me apoyaron en todo momento, siempre estaré agradecido por todo lo que han hecho por mí.

A mi hermana Lizbeth, quien siempre me impulsa a ser mejor.

A mis amigos, por todos los momentos compartidos y el apoyo recibido.

RESUMEN

A lo largo de los años, el uso de Controladores Lógicos Programables (PLCs) ha ido ganando cada vez más terreno en las diferentes industrias alrededor del mundo. Los PLCs han permitido automatizar, controlar y supervisar distintos procesos. Actualmente, se emplean varios lenguajes de programación para PLCs establecidos en el estándar IEC 61131-3, siendo el lenguaje Ladder (Escalera) uno de los más empleados.

Existen diversos softwares para programación de PLCs que permiten simulaciones, pero en el presente proyecto se pretende diseñar e implementar un software que permita poner en práctica los conocimientos acerca de PLCs y sistemas de Supervisión, Control y Adquisición de Datos (SCADA). El desarrollo del software permitirá implementar proyectos de bajo costo en un ámbito académico.

Se emplea la metodología de desarrollo de software iterativo e incremental para el desarrollo del proyecto. El trabajo solo se enfoca en los temas de automatización, supervisión y control de procesos empleando lenguaje Ladder y hardware libre como Arduino.

El software desarrollado lleva por nombre AuroraLD Studio y se compone básicamente de tres entornos de trabajo distintos: Entorno Ladder, Entorno de Tendencias y Entorno SCADA.

El entorno Ladder sirve para crear programas empleando Ladder mediante el uso de distintas instrucciones típicas de este lenguaje de programación de PLCs. Las instrucciones están basadas en los softwares de programación de los PLCs de Allen Bradley.

El entorno de tendencias permite monitorear las variables de un proceso en tiempo real mediante gráficos de tendencia. Estos gráficos de tendencias sirven para predecir o verificar el comportamiento de una determina señal. Este entorno permite incluso exportar los datos a Excel para su posterior evaluación según se requiera.

Finalmente, el entorno SCADA permite la supervisión de las variables de forma gráfica empleando distintos elementos. Estos elementos están directamente asociados a las variables creadas en el entorno Ladder por lo que el funcionamiento del SCADA está vinculado a un determinado programa Ladder que se encuentre en ejecución.

En el presente documento se detalla información acerca del desarrollo de AuroraLD Studio, así como los antecedentes y el marco teórico necesario para el entendimiento del proyecto.

Las pruebas del software se realizaron a través de simulaciones (en AuroraLD Studio) y empleando también el PLC1805 (banco de pruebas desarrollado) determinando así la utilidad del mismo. Se verificó a su vez el cumplimiento con el estándar IEC 61131-3 evaluando distintos aspectos de AuroraLD Studio.

Palabras clave: PLC, IEC 61131-3, Lenguaje Escalera, Software, SCADA, Arduino, Automatización, Supervisión, Control, AuroraLD Studio, PLC1805.

ABSTRACT

Throughout the years, the use of Programmable Logic Controllers (PLCs) has been gaining ground in different industries around the world. PLCs have allowed automation, control and supervision of different processes. Nowadays, PLCs use various programming languages established in the standard IEC 61131-3, being the Ladder language the most used.

There are many programs for PLCs that allow simulations, but the intention of this project is to design and implement a software that allows to put into practice PLC knowledge and Supervisory, Control and Data Acquisition (SCADA) systems. It is expected that the software would help students implement low-cost projects in an academic field.

The iterative and incremental model for software development is being used to develop the project. The present thesis focuses in automation, supervision and control of processes using Ladder language and open-source hardware like Arduino.

The developed software is named AuroraLD Studio and it basically comprises three different workspaces: A Ladder Workspace, a Trends Workspace and a SCADA workspace.

The Ladder workspace is use to create programs using Ladder language by adding different instructions typically found in this PLC programming language. The instructions are based on the programming software of Allen Bradley's PLCs.

The trend workspace allows to monitor variables in real time using plots. These plots are used to predict or check the behavior of a determined signal. This workspace even allows to export data to Excel making it possible to evaluate this data if needed.

Finally, the SCADA workspace allows supervision of variables graphically using different elements. These elements are directly associated with variables created in the Ladder workspace therefore its function is linked to a determine Ladder program that its being executed.

In this thesis detailed information about the development of AuroraLD Studio is provided as well as different information related to the topic making the project easy to understand.

Software tests were performed using the simulation tool (in AuroraLD Studio) and also using the PLC1805 (implemented testbed) determining the software usefulness. Compliance with the standard IEC 61131-3 was also checked evaluating different aspects of AuroraLD Studio.

Keywords: PLC, IEC 61131-3, Ladder Language, Software, SCADA, Arduino, Automation, Supervision, Control, AuroraLD Studio, PLC1805.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

INTRODUCCIÓN

El presente trabajo tiene por objetivo desarrollar un software basado en el estándar IEC 61131-3 para programación Ladder y supervisión SCADA compatible con Arduino. La implementación del software se lleva a cabo empleando el lenguaje Processing, el cual está basado en Java por lo que emplea una sintaxis similar.

Se utiliza también la programación orientada a objetos para implementar los elementos del Sistema de supervisión, control y adquisición de datos (SCADA), así como las diferentes instrucciones con las que cuenta el software.

Mediante el uso de la metodología de desarrollo de software iterativo e incremental se realiza el proyecto, dividiendo las diferentes funciones del software en actividades que se detallan en este trabajo.

El proyecto consta de cuatro capítulos que a grandes rasgos se resumen a continuación:

Capítulo 1 – Marco Metodológico detalla el planteamiento y descripción del problema, antecedentes, justificación, objetivos, alcances y limitaciones del proyecto, así como la metodología empleada.

Capítulo 2 – Marco Teórico se describe los principales conceptos y se organiza el conocimiento que se requiere conocer previo al desarrollo del proyecto. Principalmente se enfoca en el estándar IEC 61131-3, Arduino y programación orientada a objetos.

Capítulo 3 – Desarrollo del Proyecto se detalla el desarrollo del proyecto al dividir la implementación en actividades que se deben cumplir y desarrollar en base a la metodología. Se parte de las opciones más básicas y elementales para tener una interfaz de usuario y se escala progresivamente hasta desarrollar y agregar todas las funciones del software.

Capítulo 4 – Pruebas y Resultados contiene las pruebas realizadas en el software a través de un ejemplo planteado y resuelto en AuroraLD Studio asimismo se verifica el cumplimiento del estándar IEC 61131-3.

CAPÍTULO I

MARCO METODOLÓGICO

CAPÍTULO I – MARCO METODOLÓGICO

1.1. Identificación del problema

En el campo de la automatización se requiere profesionales capacitados en la programación de PLCs. El estándar IEC 61131-3 define varios lenguajes de programación que se pueden emplear, así como también las características con las que debe contar un determinado software.

La adquisición de un PLC puede resultar costoso para un estudiante si el objetivo es el aprendizaje y entendimiento de los temas relacionados a la programación de PLCs. Además, muchos softwares tienen licencias y solo es posible enlazarlos con su hardware respectivo. A nivel académico se presenta un problema al no tener suficientes equipos para los estudiantes lo cual ocasiona que no se practique lo suficiente o se ponga en práctica los conocimientos aprendidos durante las clases teóricas.

Actualmente en el mercado se encuentra placas de desarrollo como Arduino el cual permite integrar distintos componentes electrónicos y adaptarse según las necesidades de los estudiantes convirtiéndolo así en uno de los preferidos. Estas placas de desarrollo son relativamente asequibles y pueden emplearse para aprender programación Ladder, brindando así una herramienta más para los estudiantes no solo en un entorno de simulación sino también ayudando a implementar proyectos de bajo costo. Todo esto es posible si se tiene un software para trabajar con la programación Ladder y que además brinde herramientas que permitan diseñar un sistema SCADA.

1.2. Descripción del problema

Debido a los costos que supone comprar un PLC y pensando en el aprendizaje de los estudiantes, acerca de automatización, supervisión y control de procesos con PLC, se ve por conveniente desarrollar un entorno de programación Ladder.

El programa contará con instrucciones similares al software RSLogix 5000 (Rockwell Automation) y debe ser compatible con Arduino Uno y Arduino Nano. Además, el software debe permitir la supervisión mediante un sistema SCADA. La interfaz debe ser lo más sencilla posible para facilitar el aprendizaje y entendimiento de las herramientas del software. Al emplear software y hardware libre se espera una reducción de costos en la implementación de proyectos académicos.

1.3. Área de conocimiento

El presente proyecto de acuerdo a los Campos de Investigación y el Desarrollo OCDE se encuentra ubicado en:

- **Área:** Ingeniería Mecatrónica
- **Sub área:** Sistemas de Automatización, Sistemas de Control

1.4. Antecedentes

1.4.1. Antecedentes Nacionales

- "Diseño de un programador lógico programable usando microcontrolador Atmega y lenguaje Ladder para aplicaciones de laboratorio" – Universidad Nacional de Piura [1]

Ismael Sosa Yarlequé

El proyecto tiene como finalidad diseñar un PLC basado en Arduino a nivel de hardware empleando el software Waltech LadderMaker para la programación en lenguaje Ladder. Para ello se analiza las distintas señales con las que trabajará el PLC para hacer el acondicionamiento respectivo. El PLC diseñado proporciona a los estudiantes herramientas que les permita aprender acerca de la programación Ladder.

Cabe destacar que el lenguaje Ladder es uno de los lenguajes más empleado en PLC a nivel industrial. El problema principal para el autor es que los PLCs son costosos y los estudiantes no pueden adquirir uno dejando todo el estudio al respecto a nivel de simulación. Para el diseño respectivo se hace una investigación de distintos circuitos de PLCs existentes tomando aquellos que sean de información libre sin restricciones de propiedad intelectual.

El hardware implementado cuenta con: 8 entradas digitales tipo contacto con alimentación de 24v, 8 salidas digitales de tipo relé, 4 entradas analógicas de -10v a 10v, 4 salidas analógicas de 0 a 10v, comunicación USB para la programación y una fuente de alimentación de 24 voltios. Todo lo mencionado anteriormente se realiza empleando un Arduino Mega debido al número de entradas y salidas.

1.4.2. Antecedentes Internacionales

- "Design and implementation of a development environment on ladder diagram (HT-PLC) for Arduino with Ethernet connection" – IEEE – 2018 [2]

Inzunza Villagómez Héctor Iván, Pérez Arce Beatrice, Hernández Ruiz Sergio Iván, López Corella José Alejandro

Los autores en su investigación tienen como propósito presentar una alternativa de programación para las placas Arduino. Para esto se desarrolla un entorno de programación Ladder útil tanto a nivel académico como en procesos de automatización sin emplear muchos recursos. El artículo menciona que pequeñas empresas no pueden automatizar sus procesos debido al costo de un PLC. Para esto se crea un PLC basado en Arduino con conexión Ethernet denominado HT-PLC, así como un software de programación Ladder desarrollado en Visual Studio empleando el lenguaje C#. El proyecto está enfocado principalmente en el sector educativo.

El software divide el programa en una cuadrícula compuesta de 8 bloques de los cuales 7 son para las entradas y 1 para las salidas. Después de finalizado un proyecto el software permite guardar en formato de texto (.txt).

El PLC diseñado cuenta con 8 entradas digitales y 8 salidas de relé, un Arduino Uno, un Arduino Nano, 5 botones para el control y un par de luces LED como indicadores, conexión Ethernet, una entrada de 12 voltios y una pantalla LCD.

Las instrucciones con las que cuenta el software desarrollado son contactos normalmente abiertos/cerrados, salidas, líneas, temporizadores y contadores.

- "Development of Ladder Diagram Software for Learning Programmable Logic Controllers with Arduino" – Journal of Polytechnic Science – 2018 [3]

Takuya Hirata y Yoshihiro Hidaka - Japón

En el artículo se menciona que no muchas escuelas técnicas tienen acceso a suficientes PLCs debido a su precio elevado. Por esta razón se propone emplear Arduino como PLC y desarrollar un entorno de programación Ladder. La investigación se enfoca en lo académico y evalúa si el software diseñado es capaz de ayudar a comprender mejor el uso de un PLC. Se construye un módulo basado en Arduino Uno. Por otro lado, el software se programó en Microsoft Visual Studio 2015 empleando C# como lenguaje de programación.

El programa se puso a prueba y se realizó una encuesta a los estudiantes acerca del funcionamiento y manejo del software. Adicionalmente el software permite generar código entendible por Arduino que fácilmente se puede visualizar en el entorno de programación Arduino IDE.

- "Modul Programmable Logic Controller (PLC) Berbasis Arduino Severino - Módulo de Controlador Lógico programable (PLC) basado en Arduino Severino" – Journal Edukasi Elektro – 2017 [4]

Aris Susanto, Sunomo - Indonesia

En muchas industrias se requiere tener personal con competencias en el campo de control y automatización. En el artículo, el autor explica que los PLCs son costosos y no siempre la cantidad de PLCs es proporcional a los alumnos. Por eso se diseña y crea un módulo PLC basado en Arduino Severino con ATmega8 reduciendo costos. Se construye una placa y se realizan distintas pruebas para probar el funcionamiento de la misma. La programación se realiza mediante lenguaje Ladder, empleando un software de uso libre llamado LDmicro.

Cabe destacar que Arduino Severino es una placa con comunicación serial muy similar a Arduino. Sino se tiene acceso a una placa Arduino se puede construir un Arduino Severino en base a los diseños PCB y soldando los componentes a mano. Los esquemas se pueden encontrar en la página de Arduino.

En el proyecto se prueban distintas funciones que ofrece el software las cuales van desde contadores, temporizadores, registros, contactos, etc.

- "Desain Model dan Simulasi PLC-Mikrokontroler sebagai Modul Pembelajaran Berbasis PLC – Diseño y simulación de modelos de microcontroladores PLC como un módulo de aprendizaje basado en PLC" – Jurnal Teknologi Rekayasa – 2017 [5]

Qory Hidayati, Fathur Zaini Rachman, Nur Yanti, Nurwahidah Jamal, Suhaedi - Indonesia

En el estudio, se construye un entrenador PLC enfocado en el aprendizaje del mismo. Este PLC tiene como controlador a un Arduino Uno, el cual gestiona los módulos de entrada y salida. El lenguaje de programación empleado en este PLC es Ladder. La programación se realiza mediante el software LDmicro. La solución implementada facilita a los usuarios aprender sobre PLCs a un precio más accesible ya que pocas personas pueden acceder a un PLC debido al costo.

- LDmicro Software [6]

LDmicro es uno de los softwares de programación Ladder para Arduino. Este software está recomendado para el uso de OSIMPLC en entornos educativos. Asimismo, tal cual se detalla en su página web, LDmicro permite desarrollar aplicaciones simples enfocadas en un sector productivo y permite una fácil configuración de entradas/salidas.

- LadderMaker Software – Waltech Systems [7]

Por otro lado, el software LadderMaker de Waltech Systems, escrito en Python, funciona como entorno de programación Ladder con soporte para Arduino. El software permite trabajar con entradas/salidas digitales y analógicas. La última versión del software fue lanzada en 2015.

- Outseal Studio [8] - Indonesia

Uno de los PLCs basados en la placa Arduino es Outseal PLC [8]. La página web de Outseal tiene contenido acerca de cómo armar un PLC económico, así como también un software de programación llamado Outseal Studio, el cual permite la programación Ladder.

- "The Most Important Open Technologies for Design of Cost-Efficient Automation Systems" – 2019 [9]

Los autores del artículo, hacen una revisión de las diferentes soluciones que existen para automatizar procesos, donde la inversión económica es reducida empleando así software y hardware abierto. En los últimos años ha existido un incremento de hardware abierto como: Arduino y Raspberry PI, aumentando así la inclusión de estas plataformas en distintos proyectos. Actualmente, en el mercado existen soluciones industriales basadas en Arduino y Raspberry PI, siendo estas: Controllino (Arduino), Industrino (Arduino), OpenPLC y Revolution PI (Raspberry PI). Con cualquier dispositivo PLC se espera tener un lenguaje de programación que cumpla con el estándar IEC 61131-3. En los últimos años, software que cumple parcialmente con el estándar IEC 61131-3 ha sido creado. Entre estos softwares se pueden mencionar a: LDmicro, Beremiz, OpenPLC y 4diac. Se concluye que, con las soluciones actuales, es posible desarrollar un sistema de control y automatización basado solamente en software y hardware abierto. Se espera que existan más soluciones de hardware y software abierto en un futuro, debido al crecimiento que tiene Arduino y Raspberry PI.

Podemos notar que no existen muchos antecedentes nacionales respecto al desarrollo de software compatible con Arduino para programación Ladder y/o supervisión mediante un sistema SCADA.

Sin embargo, la información presentada a nivel nacional nos muestra como diseñar un PLC basado en Arduino que posteriormente podría conectarse al software desarrollado en este proyecto si tuviera soporte para Arduino Mega. Las ideas de ese proyecto se pueden tomar y adaptar a la placa Arduino Nano y Arduino Uno.

Por otro lado, en el ámbito internacional vemos más antecedentes respecto al desarrollo de software para programación Ladder compatible con Arduino. En algunos casos se ha desarrollado software propio y en otros se ha desarrollado software libre tales como: Waltech LadderMaker y LDmicro.

La información presentada resulta de gran importancia para el desarrollo del presente proyecto. Tomando como referencia algunas ideas y viendo hasta donde se ha logrado profundizar en los proyectos mencionados.

1.5. Justificación

El presente proyecto tiene por objetivo desarrollar una herramienta de uso académico, permitiendo así a estudiantes y/o docentes emplear Arduino con un software de programación Ladder y monitoreo SCADA.

1.5.1. Justificación social

Con el crecimiento de la automatización, control y supervisión de procesos a nivel industrial, se espera que los estudiantes afines conozcan acerca de PLCs. Al brindar un software de programación Ladder con instrucciones similares al software RSLogix 5000, se espera que los estudiantes posean una nueva herramienta para el manejo de la programación Ladder en forma práctica. Al mismo tiempo se espera influir en el aprendizaje de los conceptos acerca de la programación Ladder y sistemas SCADA.

1.5.2. Justificación económica

Los PLCs pueden resultar costosos en función de las necesidades. Al desarrollar un entorno de programación Ladder y supervisión mediante un sistema SCADA (compatible con Arduino), se reducen significativamente los costos de un proyecto académico. El software es libre y las placas Arduino no son muy costosas reduciendo así la inversión de comprar un PLC para realizar pruebas.

1.5.3. Justificación práctica

Actualmente, es muy difícil que los laboratorios de clases tengan PLCs para cada estudiante. El uso de una plataforma y hardware libre, podrá permitir a los estudiantes crear un módulo PLC y programarlo mediante lenguaje Ladder. Teniendo el software y hardware a su disposición, los estudiantes podrán incluso practicar desde casa. Asimismo, los estudiantes podrían optar por diseñar y crear un prototipo de PLC adaptado a sus necesidades.

1.6. Objetivos

1.6.1. Objetivo general

- Diseñar e implementar un software de programación Ladder basado en el estándar IEC 61131-3 y supervisión SCADA compatible con Arduino.

1.6.2. Objetivos específicos

- Establecer las instrucciones Ladder a usar tomando como base las del software RSLogix 5000.
- Definir los parámetros y tipos de variable de cada instrucción, así como el algoritmo a emplear para la ejecución del diagrama Ladder.
- Identificar los parámetros y animaciones necesarias para diseñar un sistema SCADA.

- Aplicar la programación orientada a objetos empleando el software Processing basado en Java para el desarrollo del proyecto.
- Evaluar el cumplimiento con el estándar IEC 61131-3 tomando como base los requerimientos establecidos en el estándar

1.7. Alcances, limitaciones y especificaciones

1.7.1. Alcances

Los alcances del proyecto son el diseño e implementación de un software para programación Ladder y supervisión mediante un sistema SCADA compatible con Arduino Uno y Arduino Nano. Adicionalmente al software y para facilitar el entendimiento del mismo se crearán manuales de uso y video tutoriales. En el presente proyecto se hará una explicación detallada de los diferentes algoritmos, procesos y metodología empleada para el desarrollo del software. El software se escribe en lenguaje Processing (basado en Java).

Además, se utilizará un módulo de pruebas basado en Arduino Nano (PLC1805) para probar la lógica y conexión con el software implementado.

El proyecto se conforma básicamente por tres módulos: un entorno Ladder, un sistema SCADA y entorno de tendencias (Gráficas o Trends). La comunicación serial entre hardware y software permite realizar simulaciones antes de cargar el programa en el microcontrolador.

El entorno SCADA cuenta con gráficos preestablecidos para diseñar un sistema de supervisión. Cada uno de los gráficos contará con parámetros y animaciones necesarias para diseñar un sistema SCADA.

El software se diseña con propósito académico y puede ser empleado para la enseñanza de programación Ladder y sistemas SCADA, así como también puede ayudar en la implementación de proyectos de bajo costo. Asimismo, se verifica el cumplimiento con algunas partes especificadas en el estándar IEC 61131-3.

1.7.2. Limitaciones

El software empleará las instrucciones más usadas en diagramas Ladder (detalladas en las especificaciones), dejando algunas para futuras versiones. No se profundiza en el tema de redes industriales y se opta por emplear el protocolo de comunicación serial ya establecido en la placa Arduino.

El software no se escribe de forma independiente a la resolución de la pantalla, teniendo este una resolución fija de 1366 x 698 pixeles. El software de programación Ladder exporta el programa realizado (proyecto) a código entendible por Arduino. La carga del programa al microcontrolador se realiza desde el entorno de programación Arduino IDE.

El presente proyecto solo consiste en el desarrollo de un software para programación Ladder, supervisión SCADA y visualización de datos a través de gráficos. El módulo construido (PLC1805) solo permite visualizar el funcionamiento demostrando así la comunicación entre el hardware de Arduino y el software implementado.

1.7.3. Especificaciones

1.7.3.1. Requisitos del sistema

Los requisitos mínimos para la ejecución del software en base a pruebas realizadas se muestran en la Tabla 1.

Tabla 1: Requisitos del sistema

Requisito	Mínimo
Memoria Ram	4 Gb
Sistema operativo (OS)	Windows o Linux (32 o 64 bits)
Almacenamiento	Aprox. 220 MB
Permiso Requerido	Administrador

1.7.3.2. Entorno Ladder

Las instrucciones Ladder que se incluirán en el software se muestran en la Tabla 2, en la cual se observa 8 categorías de instrucciones generales y en la descripción se tienen las instrucciones de manera más específica. Las abreviaciones de las instrucciones están basadas en el software RSLogix 5000.

Tabla 2: Instrucciones Ladder

Instrucciones	Descripción
Comunes	Rungs, Contactos NA/NC (XIC, XIO), Bobinas (OTE), Bobinas Latch (OTL) y Unlatch (OTU).
Temporizadores/Contadores	Temporizador TON, TOF, RTO, Contador ascendente y descendente (CTU, CTD), Bobina de RESET (RES).
Comparadores	Límites (LIM), Igual a (EQU), No igual a (NEQ), Menor que (LES), Mayor que (GRT), Menor o igual que (LEQ) y mayor o igual que (GEQ).
Funciones Matemáticas	Suma (ADD), Resta (SUB), Multiplicación (MUL), División (DIV), Raíz Cuadrada (SQR), Negación (NEG), Módulo (MOD), Valor Absoluto (ABS), Compute (CPT) y Escalamiento (SCP).
Función Mover (asignación)	Mover (MOV)
Funciones Trigonométricas	Arco coseno (ACS), Arcoseno (ASN), Arco tangente (ATN), Coseno (COS), Seno (SIN) y Tangente (TAN)
Funciones de conversión	Radianes (RAD) y Degrees (DEG)
Instrucciones especiales	Control PID discreto (PID)

Además, el software permite trabajar con Tags (Etiquetas) para cada una de las variables pudiendo así declarar variables internas (Memoria) o variables externas tanto de salida o entrada. Se tiene un límite de 100 variables en total (internas o externas).

Los temporizadores y contadores tienen asociados los bits Enable (EN) y Done (DN) así como también las variables Acumulador (ACC) y Preset (PRE).

El software no contempla instrucciones de saltos de línea (JMP), subrutinas (SBR), flancos de subida/bajada, registros, entre otros.

1.7.3.3. Entorno de Tendencias (Trends)

El entorno de tendencias (Trends) permite graficar variables analógicas y exportar los datos a Excel (csv). El número máximo de variables que se pueden graficar a la vez es 9 teniendo también un máximo de 120000 muestras antes de reiniciar toda la data. El tiempo de muestreo mínimo es de 50 ms.

1.7.3.4. Entorno SCADA

El entorno SCADA permite crear gráficos para la supervisión de un determinado proceso. Este entorno está básicamente dividido en tres partes: Elementos, parámetros y animaciones.

Los elementos son los gráficos preestablecidos y de estos se cuentan con 30 en total. A continuación, se detallan los elementos con los que cuenta el software:

- Texto, Cajas de entrada (Input), Cajas de salida (Output), Rectángulos, Trends
- Círculos, Botones, Interruptores, Líneas, Polígonos, Sliders, LEDs, Arcos
- Turbinas de viento, Ventiladores, Torres de enfriamiento, PLCs, Reloj, Imágenes
- Botellas, Motores, Bombas, SILOs/Tanques, Tuberías, Válvulas, Edificios
- Torre de transmisión, Símbolos, Medidor de variable analógica, Alarmas

Las animaciones que se implementarán en el sistema SCADA se muestran en Tabla 3. Adicionalmente a los nombres de cada una de las animaciones se tiene una pequeña descripción.

Tabla 3: Animaciones SCADA

Animación	**Descripción**
Visibilidad (Visibility)	Permite visualizar un elemento cuando se cumple una determinada condición
Parpadeo (Blink)	Permite hacer parpadear un elemento cuando se cumple una determinada condición
Mover (Move)	Permite mover un elemento cuando se cumple una determinada condición
Rotar (Rotate)	Permite rotar un elemento cuando se cumple una determinada condición
Color (Color)	Permite cambiar el color a un elemento cuando se cumple una determinada condición
Llenado (Fill)	Permite cambiar crear una animación de llenado cuando se cumple una determinada condición

Es importante resaltar que las animaciones detalladas en la Tabla 3 no son aplicables para todos los elementos del sistema SCADA.

1.7.3.5. Respecto al Hardware

Arduino será empleado para establecer la comunicación Serial con el software, esta comunicación se realizará a 115200 baudios. El software trabajará con Arduino Nano y Arduino Uno pudiendo así mejorar el algoritmo en futuras versiones para dar soporte a otras placas de Arduino. Es necesario cargar un programa previo en el microcontrolador para que este pueda trabajar con el software, cabe mencionar que esto no será necesario una vez que se desee cargar y mantener el programa de forma permanente en el microcontrolador.

Tanto si se trabaja con Arduino Nano o Arduino Uno solamente están habilitadas 6 entradas y 6 salidas digitales, así como también 4 entradas y 4 salidas analógicas.

1.7.3.6. Respecto al software

Al emplear Java el software puede correr en la mayoría de sistemas operativos. Processing permite exportar aplicaciones para Windows, Linux y Mac. Debido a que no se tiene un dispositivo MAC solo es posible exportar el software para Windows y para Linux. Adicionalmente, la interfaz del software ha sido escrita en inglés ya que es uno de los idiomas más empleados a nivel mundial. Es necesario mencionar que no es necesario usar Arduino para realizar pruebas en el software ya que tiene también una función que permite hacer simulaciones tanto en el entorno Ladder, Scada y Trends.

1.8. Metodología para el desarrollo del proyecto

El presente proyecto se desarrolla en base a la metodología de desarrollo de software iterativo e incremental, proceso que se puede observar en la Figura 1. Esta metodología es un proceso cíclico que se divide en 5 etapas. Estas etapas se realizan de forma continua y cada vez agrega y mejora las funcionalidades del software en un número indefinido de iteraciones hasta lograr el producto final [10].

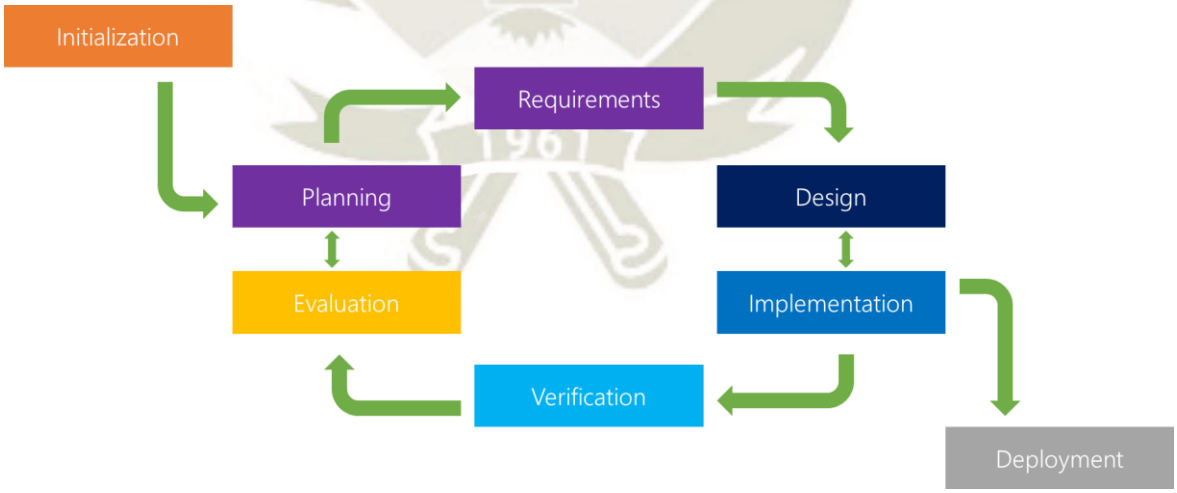


Figura 1: Modelo Iterativo [10]

Planificación y requerimientos: Se revisa bibliografía, documentación y se establecen los requerimientos de hardware y software.

La planificación y requerimientos de cada actividad siempre son diferentes en cada una de las iteraciones.

Análisis y diseño: Una vez que se termina la etapa de planificación se realiza un análisis de la lógica y modelos a emplear. El diseño hace referencia a los requerimientos técnicos necesarios para hacer lo especificado en el análisis.

Implementación: En esta etapa inicia el proceso de programación e implementación de las funcionalidades planificadas, analizadas y diseñadas.

La implementación del código se realiza en el software Processing (basado en Java) empleando la programación orientada a objetos.

Pruebas: Una vez que se ha construido una iteración del software se realizan pruebas para detectar posibles bugs o fallas en la programación.

Evaluación: Se realiza una evaluación profunda del software implementado hasta el momento y no solo de la iteración creada. Se verifica el estado del software, donde está, cuanto falta y que cosas se pueden cambiar.

Se escoge esta metodología ya que permite probar desde el primer día las funcionalidades agregadas al software reduciendo así riesgos en la programación. Se obtienen resultados periódicamente y el progreso se mide con facilidad. Las pruebas se realizan siempre en pequeñas iteraciones encontrando errores de forma sencilla.

En conclusión, el desarrollo incremental describe las alteraciones realizadas durante el diseño e implementación de cada iteración. El proyecto se divide en partes pequeñas (incremental) y se refina mediante un proceso iterativo.

1.9. Tipo de proyecto

El presente proyecto es de ingeniería aplicada y tiene por objetivo desarrollar un software de uso académico para la programación Ladder y supervisión mediante un sistema SCADA. Se realizarán pruebas a nivel de simulación y de forma práctica empleando el banco de pruebas PLC1805 v1.0 para verificar el funcionamiento del software.

CAPÍTULO II

MARCO TEÓRICO

CAPÍTULO II – MARCO TEÓRICO

2.1. Teoría básica

2.1.1. Estándar IEC 61131-3

El estándar IEC 61131 resume los requerimientos de los sistemas PLC a nivel de hardware y software [11]. Por otro lado, el estándar IEC 61131-3 define los siguientes tipos de lenguajes de programación de PLCs:

- Diagrama Ladder (LD), Diagrama de bloque de funciones (FBD)
- Lista de instrucciones (IL), Texto estructurado (ST)
- Gráfico de función secuencial (SFC)

De los lenguajes de programación mencionados 3 son del tipo gráfico y 2 son textuales. El estándar es una guía para la programación de PLCs y no debe entenderse como un conjunto de reglas rígido a seguir. Se definen también los tipos de variables, funciones y bloques de funciones básicas en los sistemas de programación de PLCs [11].

2.1.2. Unidades de Organización del Programa (POUs)

En las especificaciones del estándar IEC 61131-3 se menciona el término Unidades de Organización de Programa los cuales corresponden a las unidades pequeñas e independientes del software necesarios para la construcción de programas y proyectos [11]. Existen básicamente 3 tipos de POUs que se encuentran en los softwares de programación de PLCs los cuales se muestran en la Figura 2.

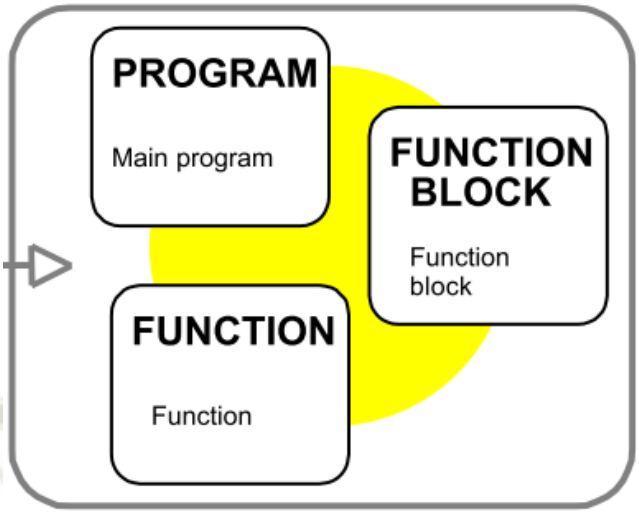


Figura 2: Unidades de Organización del Programa IEC61131-3 [11]

- **Programa (PROG):** El programa de acuerdo al estándar hace referencia al programa principal y engloba las instrucciones, variables y direcciones.
- **Función (FUN):** Las funciones siempre dan como resultado el mismo valor cuando es llamado con los mismos parámetros. Ej. ABS, ADD, MUL, etc.
- **Bloques de función (FB):** Los bloques de funciones a diferencia de las funciones no siempre dan el mismo resultado para la misma entrada y almacenan sus propios datos y pueden recordar información. Ej. TON, CTU, PID, etc.

2.1.3. Processing

Processing es un lenguaje de programación para aprender a programar dentro de un contexto de artes visuales, el entorno de desarrollo (PDE) hace que sea de fácil entendimiento. Los programas generados en Processing se denominan sketch y estos son guardados en el sketchbook [12].

Processing no cuesta nada al ser un programa libre de código abierto. Además, está escrito sobre el lenguaje de programación Java integrando muchas de las funciones de este.

El lenguaje Processing surgió en el MIT (Instituto tecnológico de Massachusetts) en el año 2001 iniciado por Casey Reas y Benjamin Fry. Processing tiene mucha aplicaciones para el desarrollo de aplicaciones web, proyectos de arte en galerías y museos [13].

2.1.4. Programación orientada a objetos

La programación orientada a objetos (POO) se basa en objetos propiamente dichos y la interacción de estos con su entorno y otros objetos. Este tipo de programación define los conceptos de objetos, clases, métodos y atributos o propiedades [14].

Los beneficios de la POO es la reutilización del código y su eficiencia en determinadas aplicaciones. Existen 4 principios de la POO: Encapsulación, Abstracción, Herencia y Polimorfismo [15].

- **Encapsulación:** Agrupar distintos elementos bajo una misma entidad o clase y colocarlos en un mismo nivel de abstracción.
- **Abstracción:** Permite incluir datos y funciones en una estructura única denominada clase [16]. Una clase es básicamente un conjunto de atributos y métodos (acciones que realizan los objetos).
- **Herencia:** En [16] especifica que de acuerdo a Luis Joyanes (1998), la herencia “Es la capacidad para crear nuevas clases de objetos que se construyen basados en clases existentes”. La clase principal se denomina clase base y las creadas a partir de esta son clases derivadas.

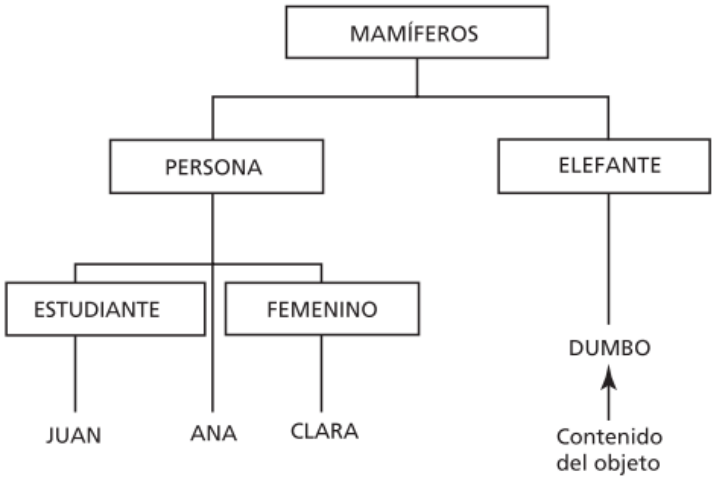


Figura 3: Herencia (POO) [16]

- **Polimorfismo:** Capacidad que tienen los objetos para responder de distintas formas ante un mismo mensaje. De esta forma se implementan distintas formas de un determinado método [16].

  Entre los lenguajes de programación que emplean POO se encuentran:

- Java, Javascript, Python, C++, Visual basic, etc.

En la Figura 4 se observan los elementos básicos de la programación orientada a objetos teniendo así una clase (humano), un objeto (nombre), propiedades (correo, dirección) y métodos (verificar y enviar mensajes).

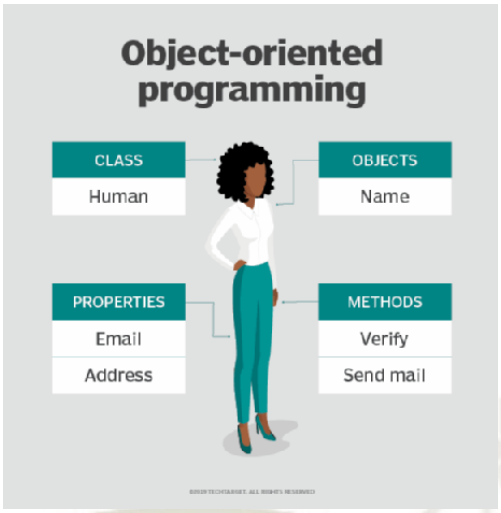


Figura 4: Programación orientada a objetos [15]

2.1.5. Diagrama Ladder

El diagrama Ladder es muy similar a los esquemas de relés lógicos que se empleaba antes de la aparición de los PLCs. Es un lenguaje de programación gráfico de PLCs establecido en el estándar IEC 61131-3 bajo el nombre Diagrama Ladder (LD) [11].

Posee instrucciones necesarias para desarrollar procesos de automatización en muchas aplicaciones, permitiendo así tener procesos secuenciales y lógicos donde se emplean contadores, temporizadores y funciones aritméticas.

Hoy en día es el lenguaje de programación de PLCs más empleado en la industria. El diagrama Ladder se compone básicamente de rieles de alimentación (verticales) a los lados y líneas horizontales de conexión (Rungs) para los diferentes componentes. Básicamente se asemeja mucho a la forma de una escalera (Ladder) [17].

En el estándar IEC 61131-3 se especifica además las condiciones de entrada Ladder que suelen ocupar el lado izquierdo del diagrama también denominado Computing y las salidas del diagrama Ladder que ocupan el lado derecho del diagrama denominado Saving tal como se muestra en la Figura 5.

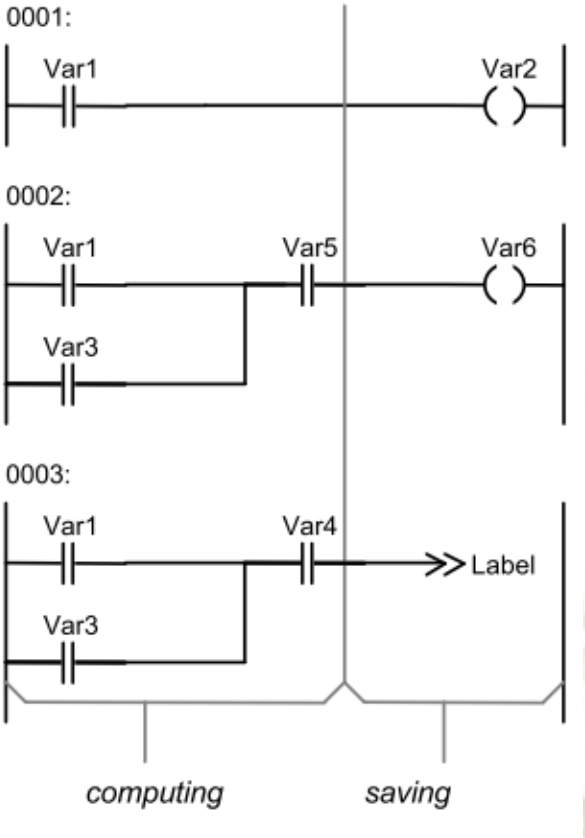


Figura 5: Estructura de un diagrama Ladder [11]

2.1.6. Sistemas SCADA

El acrónimo SCADA hace referencia a Supervisión, Control y Adquisición de datos (Supervisory control and data acquisition). El uso de este tipo de sistemas permite a la industria tener control de sus procesos de manera local o a distancia, monitorear el proceso y tener datos en tiempo real. Para asegurar un buen funcionamiento se necesitan sensores, protocolos de comunicación y PLCs o RTUs (Unidades terminales remotas). El tener control sobre un determinado proceso ayuda a los operadores a tomar mejores decisiones y tener un proceso más eficiente [18].

2.1.7. Controladores lógicos programables (PLC)

Los controladores lógicos programables (PLC) tienen múltiples aplicaciones para el control y automatización de procesos industriales. Estos dispositivos se diseñaron para

reemplazar los sistemas de relés lógicos para el control de máquinas. Se introdujeron por primera vez en el mercado a finales de 1960 siendo el MODICON 084 el primer PLC comercializado por Bedford associates [19].

Los PLCs consisten en una unidad central de procesamiento, memoria y módulos de entradas y salidas para controlar los datos del PLC. La memoria del programa es donde se aloja el programa con las instrucciones de un proyecto. La memoria de datos son los estados de las entradas y salidas (interruptores, relés, etc.). Asimismo, al PLC se conectan dispositivos de entrada y salida (hardware) [20]. En la figura Figura 6 se muestran los principales bloques que conforman un PLC.

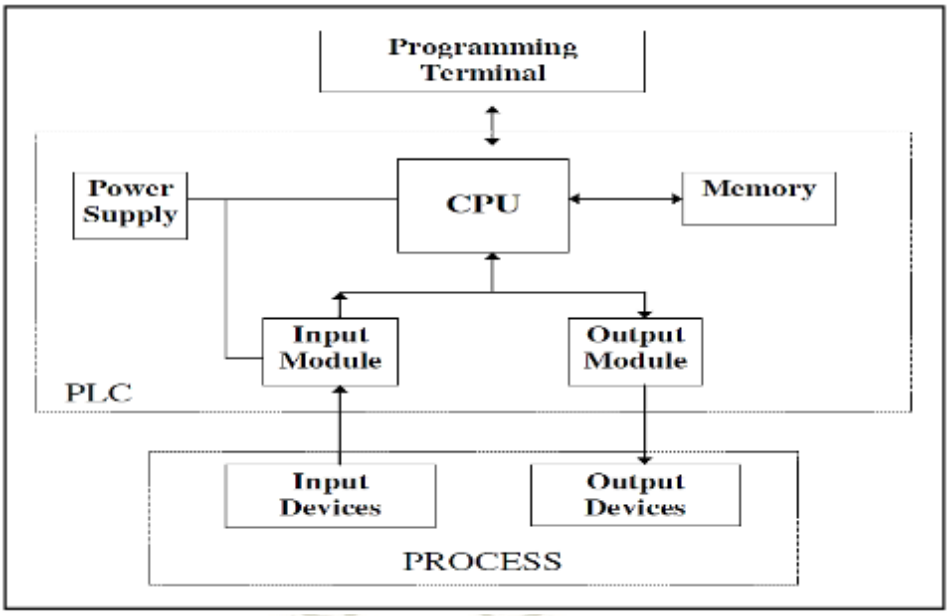


Figura 6: Diagrama de bloques de un PLC con entrada y salidas [20]

2.1.8. Arduino

Arduino es una plataforma basada en una placa electrónica de hardware libre y software de fácil uso. Existen distintas placas Arduino en las cuales se puede cargar un programa al microcontrolador a través del entorno de programación Arduino (IDE), basado en Processing. Esta plataforma se utiliza para diversos proyectos alrededor del mundo [21].

Arduino está basado en los microcontroladores AVR de Atmel: ATmega8, ATmega168, ATmega328, ATmega2560, ATmega32U4, ATmega1280, etc. Esta plataforma está orientada para ser multiplataforma y trabajar en sistemas operativos como Windows, Linux y Mac OS. En la Figura 7 se muestra un Arduino Nano basado en un ATmega328.

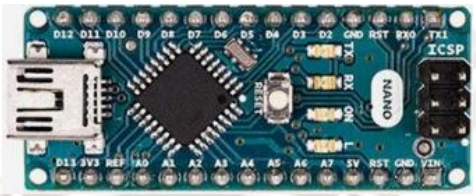


Figura 7: Arduino Nano, Obtenido de la página oficial de Arduino

En el presente proyecto se empleará Arduino Nano para la construcción del módulo PLC1805 v1.0. Las especificaciones técnicas de Arduino nano se presentan en la Tabla 4.

Tabla 4: Especificaciones de Arduino Nano, Obtenido de la página oficial de Arduino

Microcontrolador	Atmega 328
Arquitectura	AVR
Voltaje de operación	5 v
Memoria Flash	32 KB de los cuales 2KB son empleados por el bootloader
SRAM	2 KB
Velocidad de reloj	16 MHz
Pins Analógicos de entrada	8
EEPROM	1 KB
Corriente DC por I/O pin	40 mA
Voltaje de entrada	7 – 12 V
Pines digitales I/O	22 (6 PWM)
Salida PWM	6
Consumo de corriente	19 mA

2.1.9. Comunicación Serial (UART)

Se emplea el Transmisor-Receptor Asíncrono Universal (UART – Universal Asynchronous Receiver/Transmitter) para conectar dos dispositivos para establecer comunicación serial [22].

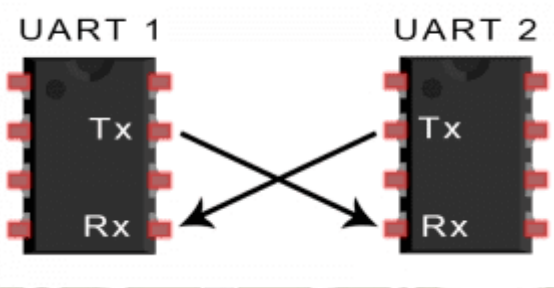


Figura 8: Conexión básica UART [22]

Los pines Tx hacen referencia al transmisor y los pines Rx hacen referencia al receptor. Cada vez que se envía un dato este es convertido en bits, estos bits se envían en paquetes con una trama de datos que involucra un bit de inicio, bits de los datos transmitidos, bit de paridad y un bit de stop. Por otro lado, cuando se recibe un dato se requieren varios paquetes para poder reconstruir la información que se recibe. En la Figura 9 se muestra la trama de datos de los paquetes al emplear comunicación serial.

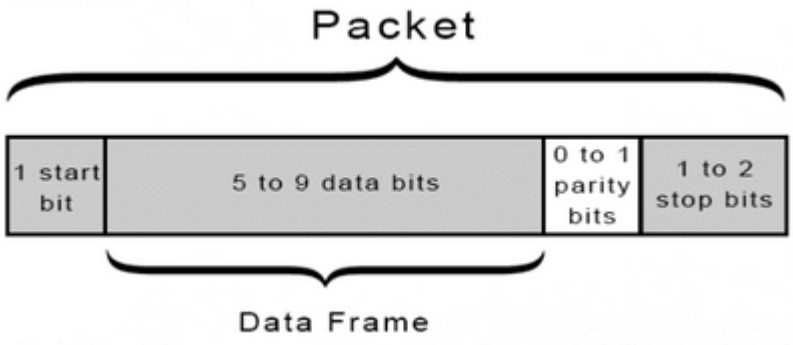


Figura 9: Trama de datos del paquete [22]

Este tipo de comunicación involucra los siguientes parámetros:

- **Baud Rate:** Hace referencia a la velocidad de transmisión de datos, es decir, cuantos bits por segundo (bps) se envían o reciben. Algunas velocidades típicas son 9600, 1200, 2400, 4800, 19200, 38400, 57600, 115200.
- **Tamaño de datos:** Número de bits por cada byte de información que se transmite.
- **Bit de paridad:** Empleado para determinar si existe pérdida de datos en la transmisión.
- **Número de bits de stop:** Indica si se tiene uno, dos o ningún bit para indicar el fin de un paquete de datos.
- **Control de flujo:** Es un método que permite reducir el riesgo de pérdida de datos al emplear caracteres especiales para el inicio y fin de la transmisión.

CAPÍTULO III

DESARROLLO DEL PROYECTO

CAPÍTULO III – DESARROLLO DEL PROYECTO

3.1. Diseño del software

3.1.1. Arquitectura de software

El software a implementar debe estar basada en una arquitectura de software para entender mejor el funcionamiento general, para ello la arquitectura que mejor encaja con lo que se pretende desarrollar es la descomposición modular. Esta arquitectura nos permite descomponer un sistema en módulos más simples que representen algo completo y que permitan codificarse de manera independiente.

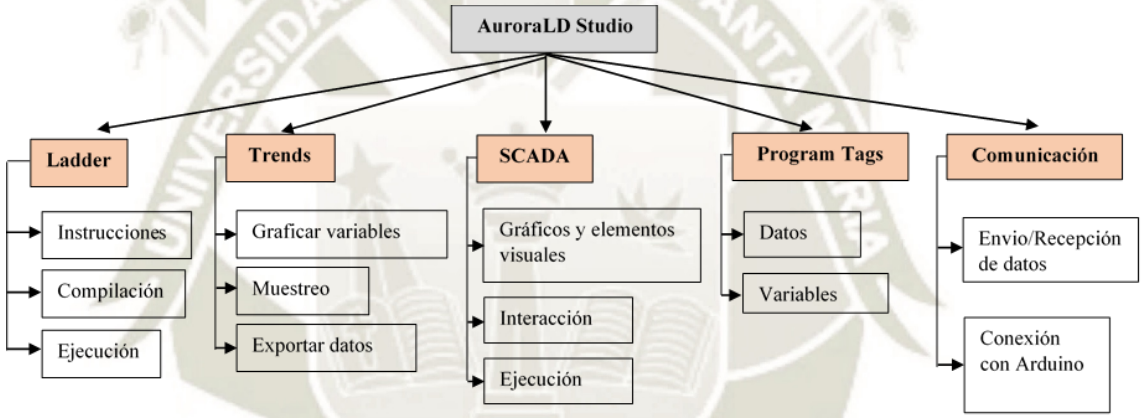


Figura 10: Arquitectura del software

La Figura 10 nos muestra los módulos en los que se ha divido el software para facilitar su implementación y pese a cada uno tener un código independiente estos comparten datos con la tabla de datos del programa. En la figura observamos también que cada módulo cumple de forma independiente con una parte específica del software.

Ninguno de los módulos detallados trabaja de forma secuencial y su funcionamiento se ejecuta de forma paralela. El módulo de comunicación es el responsable de recibir los datos y enviar datos al microcontrolador. Los datos siempre se solicitan a la tabla de datos del programa en todos los casos y es la única forma de interacción entre módulos.

3.1.2. Programación orientada a objetos

El software se compone básicamente de cuatro módulos importantes con los que interactúa el usuario: el entorno Ladder, el entorno de gráficos (trends), el entorno del sistema SCADA y la tabla de datos del programa (Program tags). Estos tres entornos deben de compartir información entre sí a través de la tabla de datos del programa.

La tabla de datos del programa o también denominado etiquetas del programa es responsable de almacenar los valores de las etiquetas declaradas en un proyecto. En todo momento de ejecución, los módulos básicos de AuroraLD Studio tienen una comunicación bidireccional a excepción del entorno de tendencias (leer datos). El entorno Ladder y SCADA se comunican con la tabla de datos del programa (Figura 11) permitiendo así leer datos de la tabla o escribir datos en la tabla según el tipo de instrucción, elemento, etc.

Figura 11: Esquema general del software

El entorno Ladder es el encargado de generar la lógica necesaria para un determinado proyecto a través de las distintas instrucciones Ladder con las que cuenta el software. Cada una de las instrucciones están definidas como clases dentro de la programación orientada a objetos.

La ejecución de un diagrama Ladder se ejecuta siempre de arriba hacia abajo y línea por línea (Rung) determinando así primero las entradas y finalmente las salidas según la lógica de entrada.

En AuroraLD Studio se tiene 40 instrucciones Ladder diferentes. Cada una de estas instrucciones se ha definido como una clase. Al definir las instrucciones como clases se tiene la ventaja de crear objetos a partir de estas clases con atributos y métodos compartidos.

Por ejemplo, la instrucción TON (temporizador on delay) para ser implementada en AuroraLD Studio se debe definir primero cuáles son los atributos y métodos propios de los temporizadores TON así como también se debe definir que atributos son privados o públicos y accesibles por otros objetos. Para realizar lo mencionado anteriormente se emplea el concepto de encapsulación y abstracción que forman parte de la programación orientada a objetos.

La encapsulación hace referencia a las variables de los objetos y si estas son de acceso público o privado, es por eso que la encapsulación está estrechamente relacionada con ocultar la información en el código para el resto del programa. Por otro lado, la abstracción define concretamente las características específicas de un objeto definiendo así los métodos y atributos que hacen único a un objeto.

Se muestra en la Figura 12 un diagrama de clases para el temporizador On Delay en la que se puede observar los atributos y métodos contemplados para esta clase.

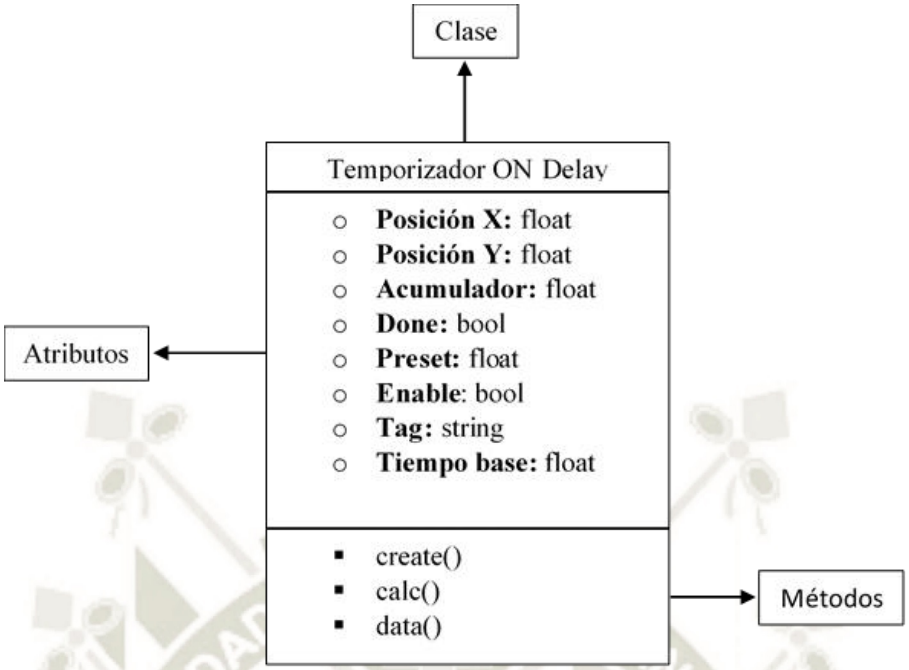


Figura 12: Diagrama de clase Temporizador On Delay

El diagrama de clases mostrado en la Figura 12 se realiza en cada una de las instrucciones determinando así los atributos y métodos que debe de tener una determinada clase. Con estos diagramas definimos y utilizamos el concepto de abstracción. En cuanto a la encapsulación podemos notar que en el temporizador On Delay los atributos de posición son privados y no necesitan ser compartidos con otros objetos ya que estos atributos permiten posicionar el diagrama de la instrucción en el espacio de trabajo Ladder empleando el método create(). Asimismo, el atributo de tiempo base también debe ser considerado como un atributo privado ya que no es necesario compartir esta información con otros objetos.

Sin embargo, los atributos del acumulador, tag, enable, preset, done, etc. son necesarios para el funcionamiento de la lógica Ladder y deben ser pasados a la tabla de datos del programa para que sean accesibles por otros entornos, objetos o elementos según sea el caso.

En base a la clase Temporizador On delay se pueden crear objetos que llevan el nombre TON con atributos ya definidos y métodos que pueden ejecutar según se requiera

durante la ejecución del software. Los objetos se denominarán TON[0], TON[1], TON[2], etc. El método calc() y create() son métodos comunes en todas las clases de instrucciones, sin embargo, al tratarse de instrucciones distintas no comparten estos métodos y son definidos de manera individual y específica en cada una de las clases implementadas.

El método data() si es común para todas las clases de instrucciones y tiene implementada la lógica necesaria que permite leer y/o escribir datos en la tabla de datos del programa de acuerdo al tipo de instrucción. Las funciones aplicables para objetos que derivan de distintas clases se conoce como polimorfismo. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al método que se ejecuta.

La manera en la que se envía los datos no siempre se ejecuta igual y depende mucho del tipo de instrucción. Algunas instrucciones envían datos enteros, booleanos, flotantes, etc. a la tabla de datos del programa y pese a no ser el mismo tipo de variable en todos los casos, los objetos responderán al método que se invoca.

En la programación orientada a objetos también está definida una característica importante denominada herencia. Cuando en el código se implementan clases que derivan de una clase mayor, pero comparten los atributos de la clase de la cual provienen hablamos de herencia. En AuroraLD Studio no se ha empleado esta característica de la programación orientada a objetos ya que se ha optado por hacer una clase para cada una de las instrucciones y/o elementos que se encuentran en el software con la finalidad de hacer más entendible el código teniendo cada una de las clases separadas a costa de una mayor cantidad de líneas de código.

El entorno de tendencias por otro lado es utilizado para tener los datos de las variables de forma gráfica con un tiempo de muestreo configurado y teniendo finalmente la

opción de exportar estos datos a Excel en un formato que separa los datos por comas (CSV). Los datos de este entorno son obtenidos de la tabla de datos del programa, es decir, que de forma indirecta obtiene los datos del entorno Ladder.

El uso de la programación orientada a objetos no tiene cabida en el entorno de tendencias puesto que solamente se emplean variables fijas (9 en total) que permiten graficar las variables analógicas o digitales que requiera el usuario. Cabe destacar que las variables que se observan en el entorno de tendencias solo son de lectura por lo que la comunicación con la tabla de datos del programa se hace en una sola dirección.

Finalmente, el entorno SCADA se constituye básicamente por los elementos predefinidos que tiene dicho entorno. Se denominan elementos a cada uno de las gráficas que se pueden agregar al espacio de trabajo SCADA. Estos elementos están asociados a las instrucciones Ladder y obtiene los valores de cada instrucción asociada de la tabla de datos del programa. Al igual que las instrucciones Ladder cada uno de los elementos del entorno SCADA están declarados como clases con sus atributos y métodos respectivos.

Por ejemplo, el elemento Botón tendría el diagrama de clases que se observa en la Figura 13.

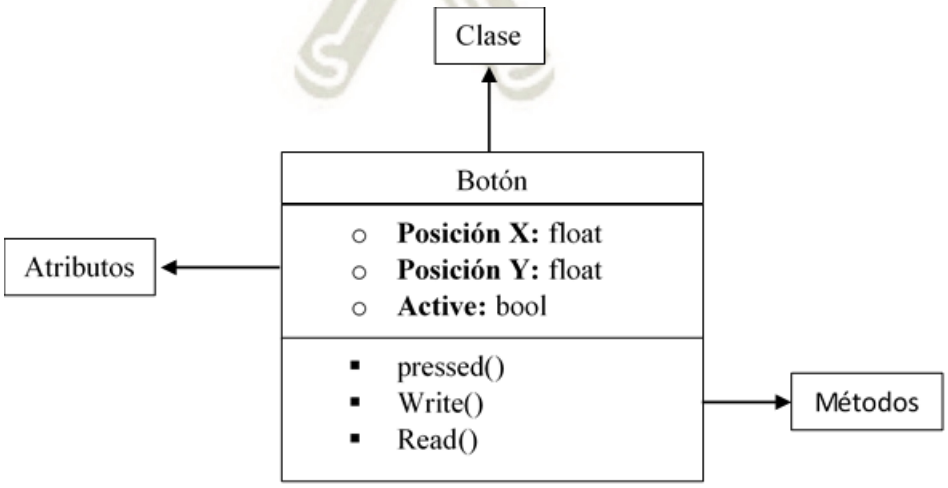


Figura 13: Diagrama de clases elemento botón

En el diagrama de clases del elemento botón observamos algunos de sus atributos y métodos posibles para la implementación de este botón. La ejecución y el comportamiento de los elementos en el entorno SCADA son muy distintos por lo que no comparten un método. Muchos de los atributos son privadas y corresponden únicamente a un objeto y define su compartimiento en el entorno. Los únicos atributos de acceso público son aquellos que van directamente a la tabla de datos del programa, entre estos atributos se encuentra: el tag, el valor del elemento ya sea digital, analógico, etc. Entre todas las funciones implementas en el entorno SCADA, la función de animación resulta importante para los elementos ya que es el único método que se comparte con todos los elementos del entorno SCADA permitiendo así generar el mismo comportamiento sin importar el tipo de elemento en la mayoría de los casos. En este entorno de trabajo no se usa el concepto de herencia ya que cada uno de los elementos ha sido declarado como una clase de la cual se derivan objetos según el usuario vaya agregando o quitando del espacio de trabajo.

La programación orientada a objetos nos ayuda a definir el comportamiento de cada una de las clases y los atributos que debe poseer una determinada instrucción o elemento antes de iniciar su implementación en el código mediante los diagramas de clases. En el caso de subclases (herencia) también es posible observar la relación que tiene una clase con otras a través de estos diagramas. Encontramos también similitudes y atributos compartidos por otros objetos y determinamos que variables deben ser vistas por el resto del programa para que permita una ejecución adecuada.

La forma de comunicación que existe entre los módulos básicos y la tabla de datos del programa es posible ya que se emplean variables globales y accesibles por todas las clases y objetos creados.

3.1.3. Comunicación de Arduino con el software

El software a implementar también permitirá la comunicación con hardware Arduino y se empleará el entorno de Arduino solamente en dos circunstancias: cargar el programa que permita la comunicación con AuroraLD Studio y cargar el diagrama Ladder al microcontrolador.

En ambos casos AuroraLD Studio permite generar el archivo necesario para establecer la comunicación con el hardware al generar código de Arduino con la extensión ino. Al usar Arduino de esta forma es la computadora quien hace los cálculos matemáticos y es la responsable de verificar la lógica Ladder en todo momento. De Arduino al computador solamente se envían las variables de entrada tanto digitales como analógicas y del computador a Arduino se envían las variables de salida tanto digitales como analógicas, todo esto mediante la comunicación serial a 115200 baudios.

En la Figura 14 se puede observar de forma gráfica el funcionamiento de Arduino cuando solo se ejecuta el código para establecer la comunicación con el software.

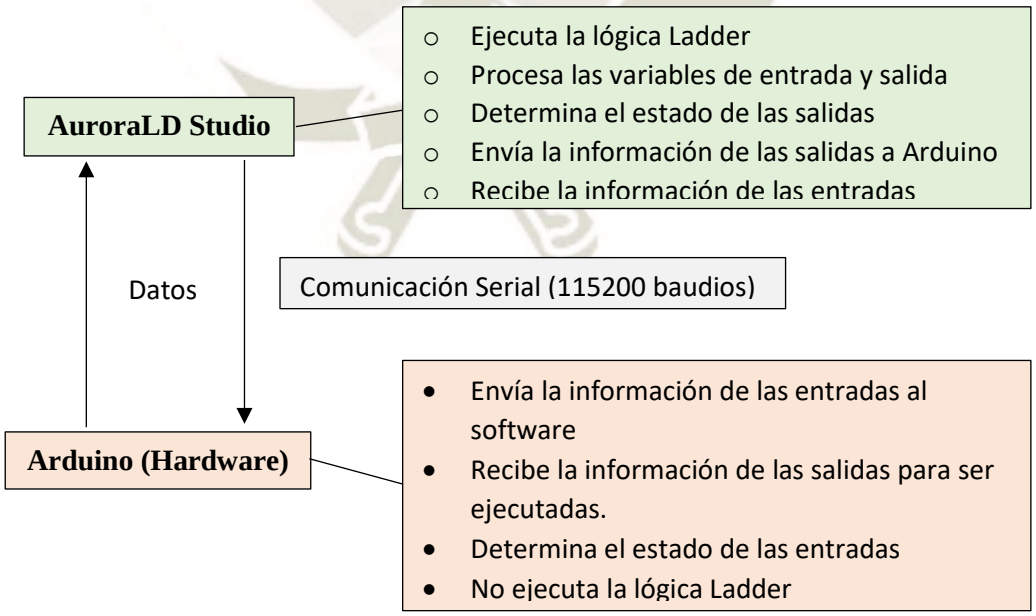


Figura 14: Arduino y AuroraLD Studio caso 1

En el segundo caso cuando el usuario lo requiera puede exportar la lógica Ladder a código entendible por Arduino con la extensión ino, convirtiendo así cada una de las instrucciones a código mediante un algoritmo implementado. El archivo generado se carga al microcontrolador a través del entorno de Arduino IDE. En este caso es Arduino quien ya ejecuta la lógica Ladder completa y no requiere del software para funcionar. El código exportado aún mantiene la comunicación serial por lo que si el usuario lo desea puede seguir trabajando con el entorno de tendencias o el entorno SCADA.

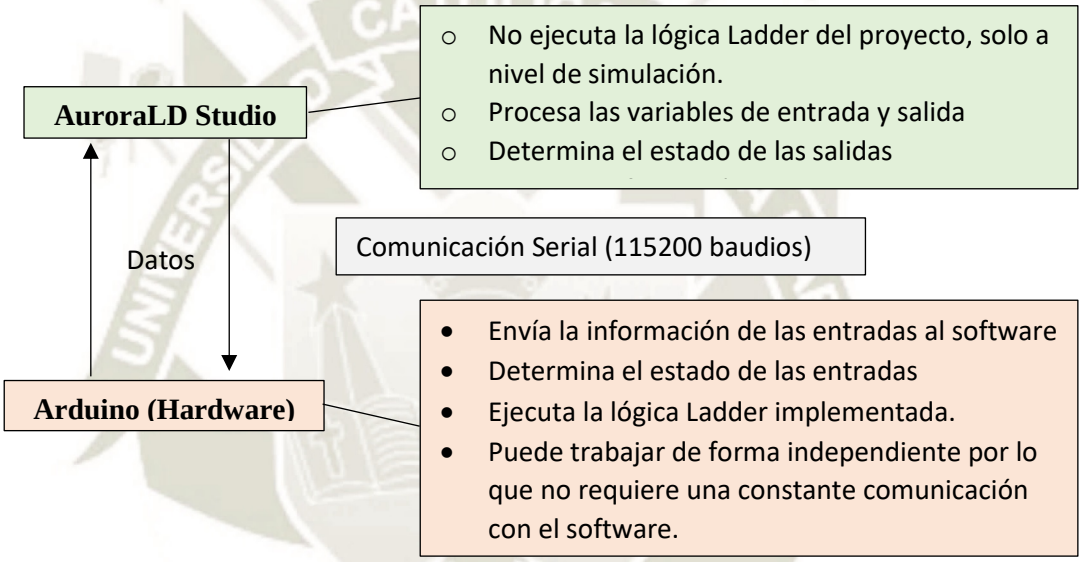


Figura 15: Comunicación entre AuroraLD Studio y Arduino caso 2

Se puede observar en la Figura 15 la comunicación que existe entre Arduino y el software implementado cuando el hardware trabaja de forma independiente y ejecuta la lógica Ladder creada en el software AuroraLD Studio.

Es importante mencionar que todo lo explicado en esta primera parte del diseño del software se detallará a mayor profundidad a medida que se construya la interfaz de usuario y se implementen las distintas clases. También se explicará la parte de la comunicación de Arduino con el software centrándose en la forma de comunicación y los datos enviados tanto por el software como por el hardware.

El desarrollo del proyecto se sustenta en el uso de la metodología de desarrollo de software iterativo e incremental. Esta metodología se empleará para el desarrollo de cada una de las partes del software.

3.2. Programación

El modo que se incluye por defecto para programar en Processing es Java. Este modo permite compilar y ejecutar un Sketch, pero cualquier edición que se realice en el código no se modificará hasta una nueva ejecución por lo que requiere detener el proyecto actual y cargarlo de nuevo.

Para evitar corregir y ejecutar el código repetidas veces se empleará otro modo denominado REPL Mode el cual permite realizar modificaciones cuando el Sketch está en ejecución, lo que se conoce como Hot Swapping. El modo en mención se instala directamente del IDE de Processing y sigue empleando Java.

Entre más funciones se empiezan a agregar al proyecto se incrementa también el número de archivos del proyecto y Processing no tiene una forma eficaz y rápida para poder gestionar los distintos archivos del proyecto dificultando así el desarrollo. Este inconveniente se soluciona al emplear un editor de texto como Atom y agregando los paquetes necesarios para poder seguir trabajando con el lenguaje de Processing. De hecho, no es que uno reemplace al otro porque se perdería la ventaja del modo REPL en Processing por esa razón lo recomendado es emplear ambos programas a la vez bajo la siguiente lógica:

- Ejecutar el código desde Processing empleando el modo REPL
- Realizar los cambios en Atom y observar los resultados

La ventaja de Atom es que permite navegar por todos los archivos del proyecto con mayor facilidad pudiendo así saltar entre una función y otra e incluso observar varios archivos a la vez para hacer las correcciones y/o implementaciones respectivas.

3.3. Interfaz del entorno Ladder

El programa a desarrollar al igual que cualquier otro software que se emplea día a día debe tener una interfaz de fácil entendimiento y contar con las herramientas necesarias que permitan la interacción con el usuario.

El único idioma con el que cuenta la interfaz de usuario será inglés ya que se considera uno de los idiomas más empleados en todo el mundo facilitando así el uso del software a estudiantes de otros países.

Luego de analizar los distintos programas de uso cotidiano podemos notar que en su barra de herramientas hay dos opciones que nunca faltan, estas son: File (Archivo) y Help (Ayuda). La barra de herramientas del software debe contar adicionalmente con opciones que permitan la comunicación con el hardware, compilación, ejecución, etc.

La distribución de las diferentes ventanas en el programa es importante ya que ayudan a tener un software ordenado. Asimismo, también se considera importante definir colores e íonos para tener un entorno atractivo.

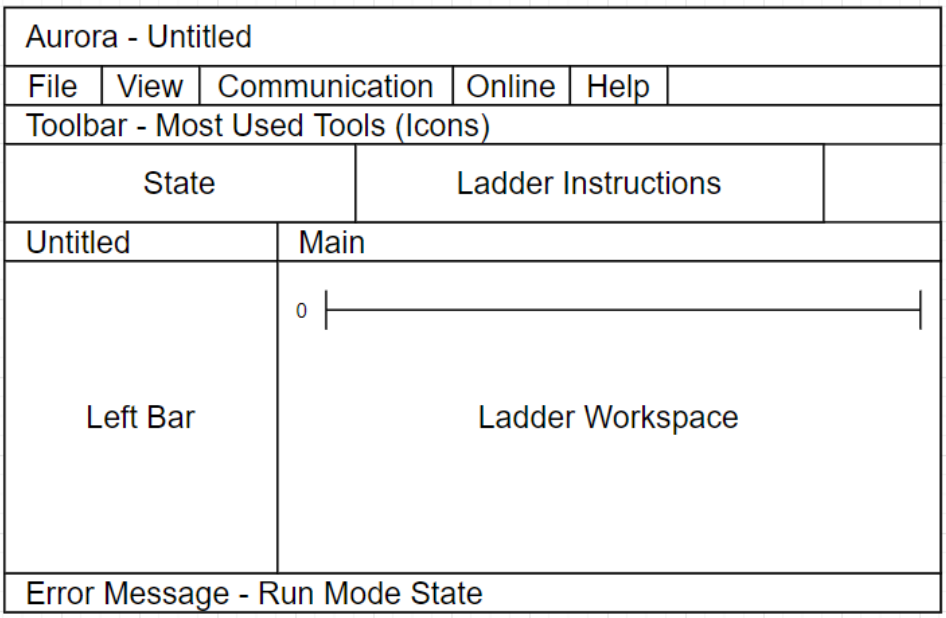


Figura 16: Diseño Preliminar AuroraLD Studio

En la Figura 16, se observa un diseño preliminar del entorno del software en el cual se detallan las ventanas, opciones y herramientas principales, así como la ubicación de las mismas.

Teniendo la distribución de las opciones en el programa es necesario definir las dimensiones, colores y como las opciones interactúan con el usuario. En este apartado se explicará el desarrollo e implementación del menú del programa, la barra de herramientas, el menú de estados y las pestañas para colocar las instrucciones Ladder.

Es importante mencionar que AuroraLD Studio está diseñado en base a una resolución fija de 1366 x 698 pixeles.

3.3.1. Colores de AuroraLD Studio

Los colores seleccionados para AuroraLD Studio son principalmente tonos de azul, plomo, blanco y verde. El modo de trabajo para definir los colores en Processing está en base a colores RGB por lo tanto se requieren 3 parámetros para definir los colores: cantidad de rojo (máx. 255), cantidad de verde (máx. 255) y cantidad de azul (máx. 255). En algunas ocasiones se empleará un cuarto parámetro para definir transparencia.

En Processing los colores se manejan en variables denominadas color. En la Figura 17 se presentan algunos ejemplos de variables de color definidas en el código fuente.

```
color off = color(255, 0, 0);
color on = color(0, 255, 0);

//COLORES
color lines = color(0, 0, 255);
color activate = color(0, 255, 0);
color grid_indicator_on=color(0, 222, 73);
color grid_indicator_off=color(170, 170, 170);
color info_color=color(0, 0, 0);
```

Figura 17: Ejemplo de Colores definidos en AuroraLD Studio

3.3.2. Coordenadas en Processing

Las dimensiones que se manejan en Processing están en base a pixeles teniendo estas coordenadas X e Y. La coordenada (0, 0) se ubica en la parte superior izquierda desde la cual se definen todas las posiciones para los elementos en AuroraLD Studio.

3.3.3. Program Menu (Menú del Programa)

El desarrollo del menú del programa incluye las opciones principales del software (File, View, Communication, Online y Help), adicionalmente a las opciones también se ha agregado la hora y la fecha. En la Figura 18 tenemos la barra del menú del programa implementada con las opciones previamente descritas.

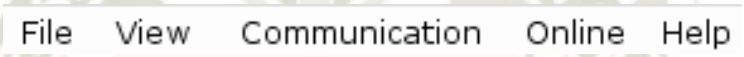


Figura 18: Menú del Programa AuroraLD Studio

En el código es necesario definir cada una de las opciones como texto de manera que se puedan mostrar cuando se requiera. Se dividen las opciones en categorías y se almacenan en Arrays de String tal y como se observa en la Figura 19.

```
//OPTIONS
String option="NONE";
String[] opt = {"FILE", "VIEW", "COM", "ONLINE", "HELP", "NONE"};
String[] file = {"New Project", "Open", "Save", "Save as", "Export", "Exit"};
String[] view = {"Program Tags", "Controller Properties", "Trends", "Scada", "Ladder"};
String[] com = {"Arduino Nano", "Arduino Uno", "Arduino Mega", "Connect", "Disconnect"};
String[] online = {"Run", "Stop", "Compile"};
String[] help = {"About Aurora", "Tutorials", "Ladder", "Trends", "Scada"};
int num_option=0;
int x=0;
```

Figura 19: Variables Program Menu

La barra con las opciones principales se ha implementado en base a rectángulos y texto. Para facilitar la programación del menú se crea una función denominada PRG_BAR. Se presenta en la Figura 20 el código que dibuja el menú del programa. En el código observamos una iteración de máximo 5 ya que son solamente 5 opciones. En la parte de abajo observamos el uso de la función text para colocar texto en pantalla con una posición X Y.

```
for(int i=0;i<5;i++){
//fill(201,228,207,c_[i]);
fill(mtop_select,c_[i]);
rect(t[i],1,t[i+1],20);
}

//TEXT
fill(mtop_text);
text("File",20,11);
text("View",62,11);
text("Communication",144,11);
text("Online",229,11);
text("Help",277,11);
```

Figura 20: Implementación Program Menu

La interacción de este menú con el usuario se realizó empleando eventos del mouse los cuales permiten reconocer Clicks del usuario, movimiento o arrastre. También se empleó las variables mouseX y mouseY para conocer las posiciones del cursor.

Dentro de cada opción principal existen opciones adicionales que se deben de mostrar cuando el usuario clickea en una de las opciones principales en el menú del programa. Para facilitar la programación y uso de las opciones adicionales se crea una función denominada OPT. Parte de la función OPT se muestra en la Figura 21.

```
if (option_.equals(opt[0])) {
  fill(mtop_option_text);
  for (int i=0; i < file.length; i++) {
    text(file[i], 9, 30+20*i);
  }
} else if (option_.equals(opt[1])) {
  for (int i=0; i < view.length; i++) {
    text(view[i], 47, 30+20*i);
  }
} else if (option_.equals(opt[2])) {
  for (int i=0; i < com.length; i++) {
    text(com[i], 97, 30+20*i);
  }
  image(check, 125+x, 21+select_board, 19, 19);
} else if (option_.equals(opt[3])) {
  for (int i=0; i < online.length; i++) {
    text(online[i], 210, 30+20*i);
  }
} else if (option_.equals(opt[4])) {
  for (int i=0; i < help.length; i++) {
    text(help[i], 264, 30+20*i);
  }
}
```

Figura 21: Opciones Adicionales Program Menu

Básicamente se comprueba la opción principal escogida y se despliega el array correspondiente a File, View, etc. (Figura 19).

Tras clickear en File se muestran las opciones adicionales: New Project, Open, Save, Save as, Export y Exit tal como se puede observar en la Figura 22.

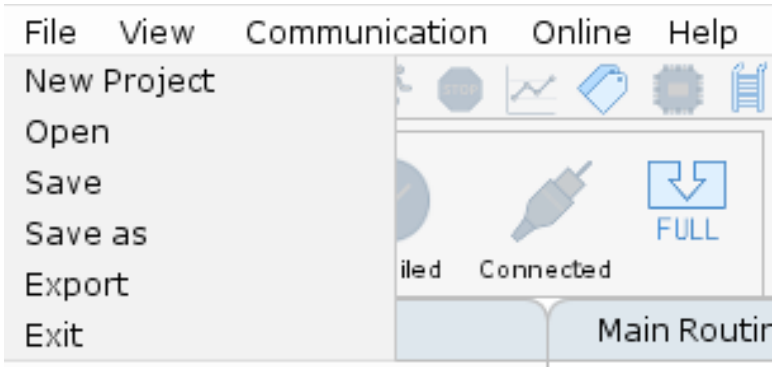


Figura 22: Opción FILE Implementada

La Figura 23 corresponde a la opción View, la cual contiene opciones correspondientes a los diferentes entornos en AuroraLD Studio, así como también información del controlador (PLC).

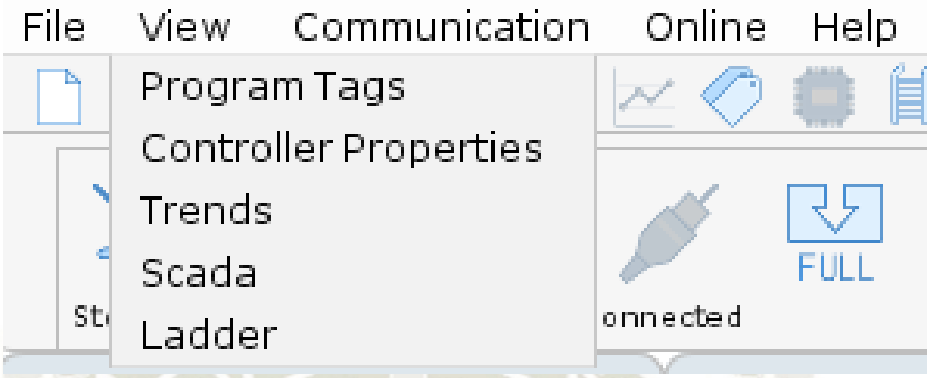


Figura 23: Opción VIEW Implementada

- **Program Tags:** Los tags del programa permiten definir las variables que se usarán en el proyecto. Se componen básicamente de 5 campos: nombre, tipo de variable, dirección, valor y descripción.
- **Controller Propierties:** Las propiedades del controlador nos permiten visualizar los pines definidos para cada uno de los microcontroladores (Arduino Nano y Arduino Uno) con las direcciones empleadas en AuroraLD Studio.
- **Trends:** Permite abrir la ventana para graficar las variables en estudio.
- **Scada:** Abre la ventana para desarrollar el entorno SCADA.
- **Ladder:** Permite visualizar el diagrama Ladder.

Por otro lado, se implementa la opción Communication la cual permite la selección del controlador y ayuda a establecer la comunicación serial con el dispositivo. La opción communication implementada se muestra en la Figura 24.

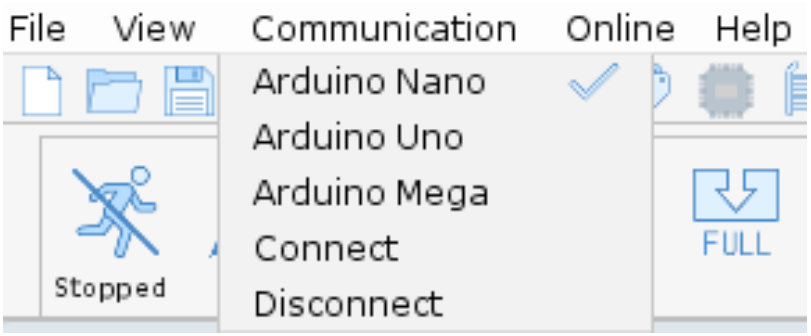


Figura 24: Opción COMMUNICATION Implementada

Esta opción nos permite realizar la comunicación con el hardware. Se definen inicialmente 3 microcontroladores de los cuales solo serán implementados 2 siendo estos Arduino Nano y Arduino Uno. La idea es seleccionar el microcontrolador a emplear y posteriormente usar la opción Connect (Conectar) para establecer una conexión con el dispositivo seleccionado. Adicionalmente se implementará una opción para terminar la comunicación con el hardware denominada Disconnect (Desconectar).

También resulta importante la opción Online ya que tiene el propósito de controlar la ejecución de un programa. Las opciones adicionales con las que cuenta Online son: Run, Stop y Compile (Figura 25). Esta opción permite poner en ejecución AuroraLD Studio y empezar a recibir y/o mandar datos al microcontrolador. Se ejecuta el proyecto al presionar Run (Ejecutar), se detiene al presionar Stop y se verifica que no exista errores en el programa antes de la ejecución presionando Compile (Compilar).

Figura 25: Opción ONLINE Implementada

Las opciones adicionales deben satisfacer las siguientes reglas cuando entren en funcionamiento:

- Si **Compile** es **Verdadero** entonces **Run** se **Habilita**
- Si **Run** es **Verdadero** entonces **Compile** se **Deshabilita**
- Si **Run** es **Verdadero** entonces **Stop** se **Habilita**
- Si **Stop** es **Verdadero** entonces **Compile** se **Habilita**
- Si **Stop** es **Verdadero** entonces **Run** se **Deshabilita**
- Si **Run** es **Verdadero** entonces **Run** se **Deshabilita**

Las reglas presentadas para las opciones adicionales garantizan un funcionamiento adecuado de las 3 opciones. Por ejemplo, no es necesario correr el programa cuando está en ejecución y tampoco es necesario detenerlo cuando está detenido.

Por último, se implementa la opción HELP, la cual contiene toda la información de ayuda necesaria para comprender el manejo del software. Las opciones se muestran en la Figura 26.

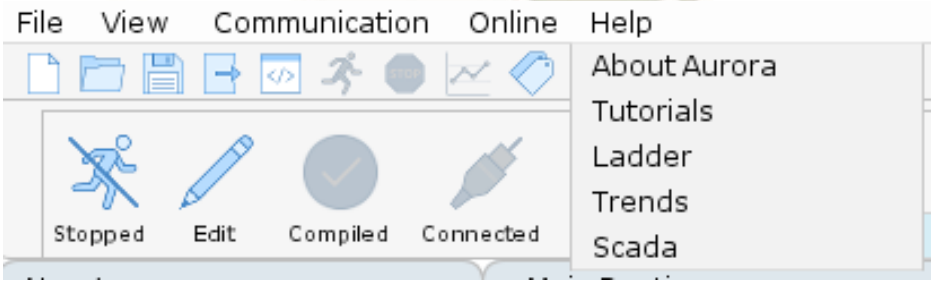


Figura 26: Opción HELP Implementada

La opción adicional About Aurora muestra la información correspondiente al software e información adicional. Esta información acerca del software se observa a detalle en la Figura 27.

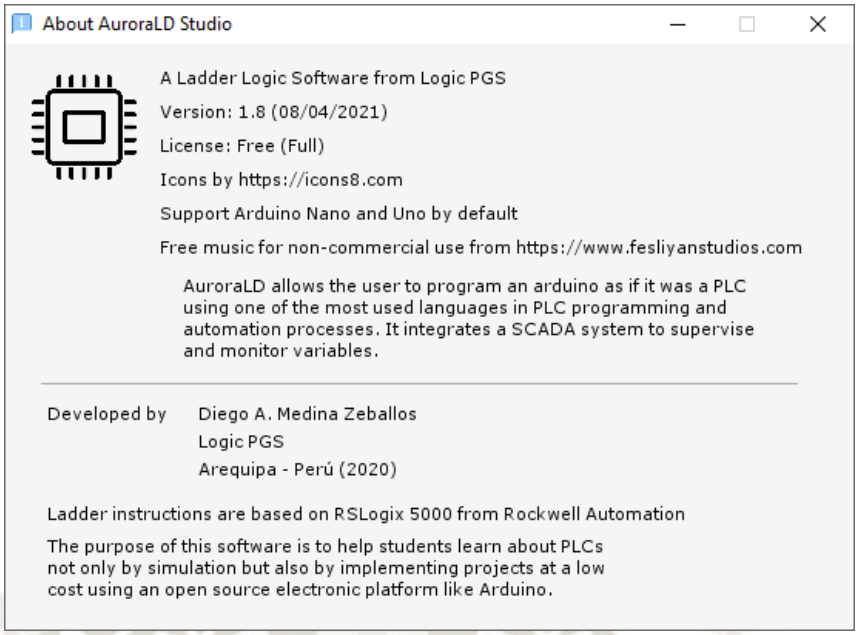


Figura 27: About AuroraLD Studio

La opción Tutorials abre un canal de Youtube cuyo nombre es LogicPGS. Es en este canal donde se tendrá videotutoriales explicando así ejemplos básicos e información importante que ayude a comprender el manejo del software. En la Figura 28 se muestra el canal de youtube y algunos de los videos que se han subido a la plataforma.

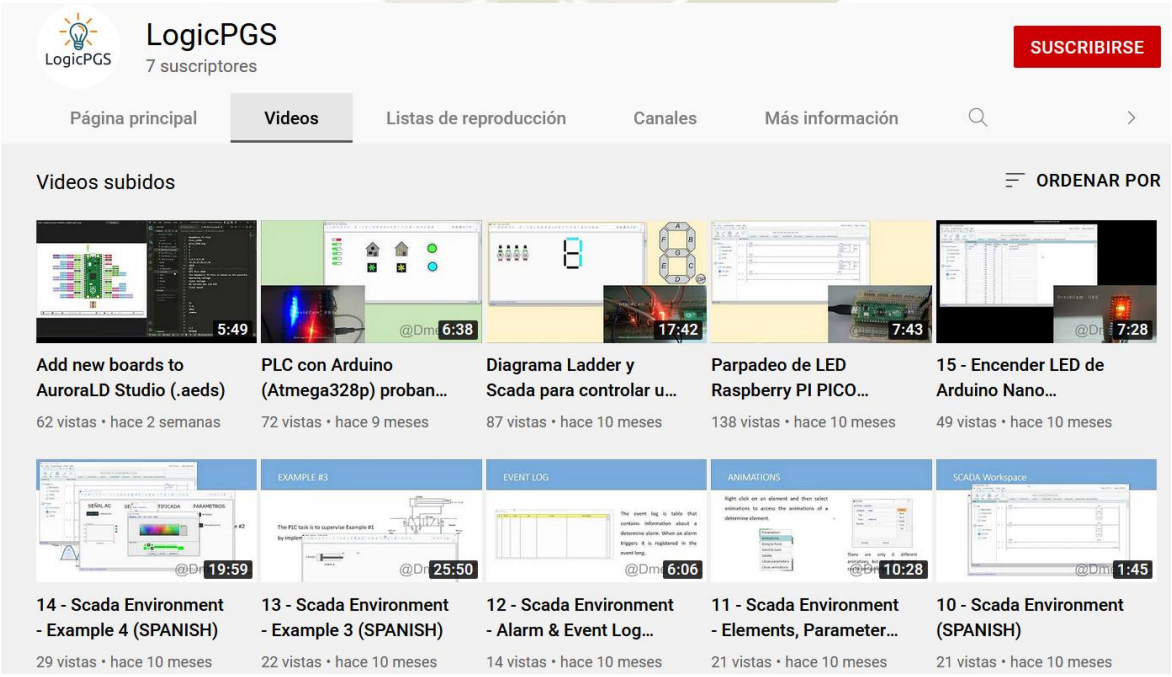


Figura 28: Canal LogicPGS

Se integra también al software 4 manuales: UIM001, UIM002, UIM003 y UIM004. Se pueden acceder a estos manuales ingresando a la carpeta de instalación o también usando las opciones adicionales de HELP: Ladder, Trends y SCADA mostradas en la Figura 26.

- UIM001 – Hace referencia a la instalación del software
- UIM002 – Hace referencia al entorno Ladder
- UIM003 – Hace referencia al entorno Trends
- UIM004 – Hace referencia al entorno SCADA

En la Figura 29 se muestra la carátula del manual UIM#002. Este manual tiene aproximadamente 98 páginas completamente en inglés en las que se explica el uso del entorno Ladder.

Los archivos con los manuales, el software y el link correspondiente al canal de YouTube se encuentran en los anexos (ver ANEXOS)

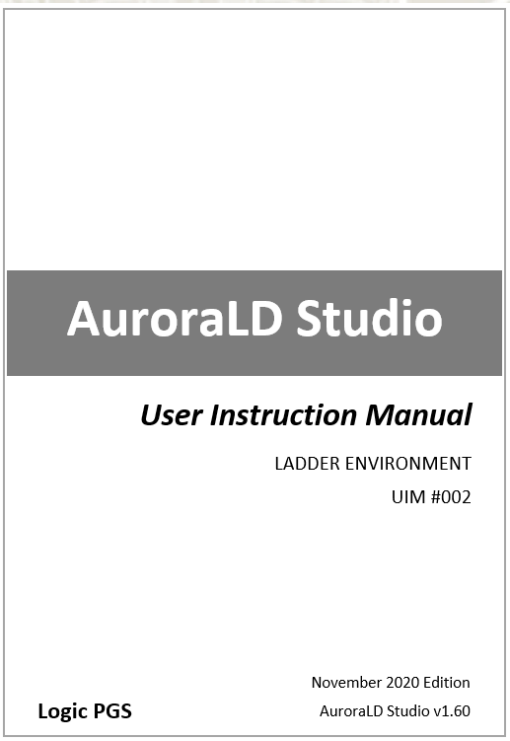


Figura 29: UIM002 Ladder Environment

Como se explicó previamente, la interacción de estas opciones con el usuario se realiza a través de eventos. En este caso se hace uso de la función mousePressed(), evento que se ejecuta una vez cada vez que se hace click en el programa.

En el interior de la función mousePressed() se ha implementado el código mostrado en la Figura 30.

```
//INDEPENDENT BUTTONS
if (parameter_window==false) {
  for (int i=0; i<num_option; i++) {

    if ((mouseX>x)&&(mouseX<155+x)&&(mouseY>20*i+20.5)&&(mouseY<20*i+40.5)) {
      if (option.equals("FILE")) {
        FILE(file[i]);
      } else if (option.equals("VIEW")) {
        VIEW(view[i]);
      } else if (option.equals("COM")) {
        COM(com[i]);
      } else if (option.equals("ONLINE")) {
        if ((i==2)&&(execution==true)) {
        } else if ((i==0)&&(compiled==false)||(i==0)&&(connected==false)) {
          if (connected==false&&project_created==true) {
            showMessageDialog(frame, "Connection has not been established", "Device not found", ERROR_MESSAGE);
          }
        } else if ((i==1)&&(execution==false)) {
        } else {
          ONLINE(online[i]);
        }
      } else if (option.equals("HELP")) {
        HELP(help[i]);
      }
    }
  }
}
```

Figura 30: Program Menu evento mouse

El código presentado en la Figura 30 permite ejecutar cada una de las opciones adicionales. Se utiliza el reconocimiento de la posición del cursor en base a la interfaz del software para determinar donde el usuario hace click y en función de eso ejecutar la opción.

3.3.4. Toolbar Options (Opciones de la barra de herramientas)

La barra de herramientas (Toolbar) está localizado en la parte superior de AuroraLD Studio y contiene las opciones más empleadas en forma de íconos. Al estar trabajando con tonos de plomo (para la interfaz) se opta por escoger íconos Azules.

Los íconos no serán diseñados, pero si descargados de Internet de la página web Icons8 (Figura 31). La página en mención permite la descarga gratuita de distintos íconos tras registrarse, la única condición es atribuir a los creadores de dichos íconos.

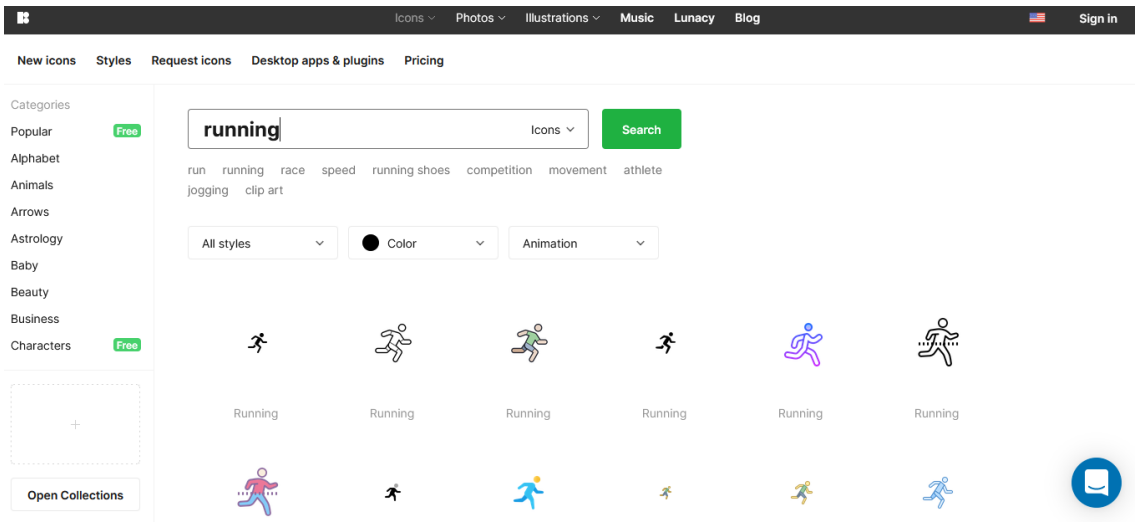


Figura 31: Página Web Icons8

Entre las herramientas más empleadas se considera: New Project, Open, Save, Export, Compile, Run, Stop, Trends, Tags, Controller Properties y Ladder. Dichas opciones ya se mencionaron con anterioridad (pág. 45).

En la Tabla 5 se muestran los íconos escogidos, así como los shortcuts correspondientes para acceder a cada una de las herramientas con mayor facilidad.

Tabla 5: Íconos y shortcuts de la barra de herramientas

N	Ícono	Herramientas	Shortcut
1		New Project	*(Ctrl + N)*
2		Open	*(Ctrl + O)*
3		Save	*(Ctrl + S)*
4		Export	*(Ctrl + E)*
5		Compile	*(Ctrl + C)*
6		Run	*(Ctrl + R)*
7		Stop	*(Ctrl + T)*
8		Trends	*(Alt + 3)*
9		Tags	*(Alt + 2)*
10		Controller Properties	*Ninguno*
11		Ladder	*(Alt + 1)*

Para poder importar los íconos en Processing estos deben guardarse en una carpeta llamada data y esta carpeta debe estar en el mismo directorio que el ejecutable del software. Todos los íconos descargados están en formato .png a 40px u 80px y guardados en el folder especificado con un nombre representativo.

Se puede apreciar en la Figura 32 una parte del contenido de la carpeta data, esta carpeta contiene todos los archivos necesarios para el correcto funcionamiento de AuroraLD Studio.

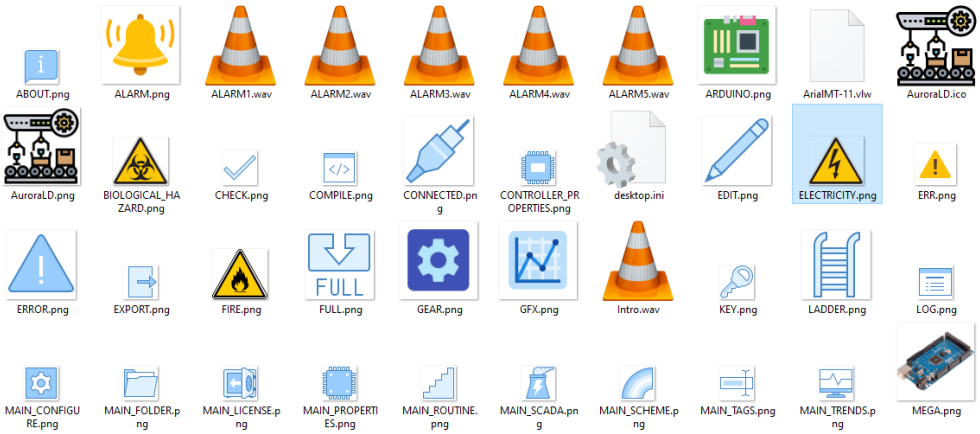


Figura 32: Carpeta Data AuroraLD Studio

Una vez que se ha creado la carpeta y se tienen los íconos correspondientes se procede a importar los íconos en Processing. Para esto se emplea la variable PImage la cual sirve para declarar una variable del tipo imagen (Figura 33), la función loadImage que permite cargar una imagen al Sketch (Figura 34) y por último image que permite mostrar la imagen en el Sketch.

```
//ICONS
PImage new_file;
PImage open;
PImage save;
PImage export;
PImage compile;
PImage run;
PImage stop;
PImage trends;
PImage tags;
PImage properties;
PImage check;
PImage ladder;
```

Figura 33: Declaración de variables PImage

```
//ICON TOOLBAR
new_file = loadImage("NEW_PROJECT.png");
open = loadImage("OPEN.png");
save = loadImage("SAVE.png");
export = loadImage("EXPORT.png");
compile = loadImage("COMPILE.png");
run = loadImage("RUN.png");
stop = loadImage("STOP.png");
trends = loadImage("TRENDS.png");
tags = loadImage("TAGS.png");
properties = loadImage("CONTROLLER_PROPERTIES.png");
check = loadImage("CHECK.png");
ladder = loadImage("LADDER.png");
```

Figura 34: Carga de Imágenes al Sketch

Para mostrar los íconos o imágenes cargadas en Processing es necesario emplear la función Image. En el caso de la barra de herramientas se crea una función llamada TOOL_BAR, la cual al ser llamada mostrará los íconos en el lugar que corresponde.

El código mostrado en la Figura 35 muestra una parte importante de la función TOOL_BAR y contiene principalmente condiciones (If) que permiten desactivar o activar determinadas herramientas.

```
for (int i=0; i<11; i++) {

  if (num_option==0) {
    if ((mouseX>rect_icon[(2*i)])&&(mouseX<rect_icon[(2*i)+1])&&(mouseY>22)&&(mouseY<48)) {
      d[i]=255;
    }
  }
  if ((i==0)&&(execution==true)) {
    tint(150, 100);
  } else if ((i==1)&&(execution==true)) {
    tint(150, 100);
  } else if ((i==2)&&(project_created==false)||(i==2)&&(execution==true)) {
    tint(150, 100);
  } else if ((i==3)&&(project_created==false)||(i==3)&&(execution==true)) {
    tint(150, 100);
  } else if ((i==7)&&(project_created==false)||(i==7)&&(connected==false)) {
    tint(150, 100);
  } else if ((i==8)&&(project_created==false)) {
    tint(150, 100);
  } else if ((i==9)&&(project_created==false)||(i==9)&&(connected==false)) {
    tint(150, 100);
  } else if ((i==10)&&(project_created==false)) {
    tint(150, 100);
  } else if (((i==4)&&(execution==true))||((i==4)&&(project_created==false))) {
    tint(150, 100);
  } else if ((i==5)&&(compiled==false)||(i==5)&&(connected==false)) {
    tint(150, 100);
  } else if ((i==6)&&(execution==false)) {
    tint(150, 100);
  } else {
    noTint();
    stroke(mtoolbar_select, d[i]);
    rect(rect_icon[2*i], 22, rect_icon[(2*i)+1], 45);
  }

  if (i==10) {
    image(icons[i], icon_pos[i], 23, 20, 20);
  } else {
    image(icons[i], icon_pos[i], 24, 20, 20);
  }
}
```

Figura 35: Función TOOL_BAR

Estas condiciones existen para habilitar o no algunos íconos en función del estado del programa, es decir, proyecto creado, en ejecución, compilado, conectado, etc. Si las condiciones se satisfacen se inhabilita una determinada herramienta mostrándose en color plomo. Por otro lado, si la condición no se satisface el ícono muestra su verdadero color. El cambio de color del ícono se realiza empleando la función Tint.

Las condiciones que se deben satisfacer para inhabilitar los íconos y que están implementadas en la Figura 35 son:

- Si **Ícono 1 (New Project)** y **Ejecución** son **Verdadero** el ícono 1 se **Inhabilita**
- Si **Ícono 2 (Open) y Ejecución** son **Verdadero** el ícono 2 se **Inhabilita**
- Si **Icono 3 (Save) y (Proyecto Creado** es **Falso** o **Ejecución** es **Verdadero)** el ícono 3 se **Inhabilita**
- Si **Icono 4 (Export) y (Proyecto Creado** es **Falso** o **Ejecución** es **Verdadero)** el ícono 4 se **Inhabilita**
- Si **Icono 5 (Compile) y (Proyecto Creado** es **Falso** o **Ejecución** es **Verdadero)** el ícono 5 se **Inhabilita**
- Si **Icono 6 (Run) y (Compilado** es **Falso** o **Conectado** es **Falso)** el ícono 6 se **Inhabilita**
- Si **Icono 7 (Stop) y (Ejecución** es **Falso)** el ícono 7 se **Inhabilita**
- Si **Icono 8 (Trends) y (Proyecto Creado** es **Falso** o **Conectado** es **Falso)** el ícono 8 se **Inhabilita**
- Si **Icono 9 (Tags) y Proyecto Creado** es **Falso** el ícono 9 se **Inhabilita**
- Si **Icono 10 (Controller Properties) y (Proyecto Creado** es **Falso** o **Conectado** es **Falso)** el ícono 10 se **Inhabilita**
- Si **Icono 11 (Ladder) y Proyecto Creado** es **Falso** el ícono 11 se **Inhabilita**

Las condiciones mencionadas existen con el propósito de impedir realizar ciertas acciones cuando otras que no guardan relación están en ejecución. Por ejemplo, ejecutar el programa e intentar crear un nuevo proyecto, compilar el programa cuando está en ejecución, etc. La barra de herramientas (toolbar) implementada se muestra en la Figura 36.

Figura 36: Barra de Herramientas AuroraLD Studio

La interacción con el usuario se realiza empleando mouseClicked, evento que detecta clicks en el Sketch. Al clickear en las herramientas se activan las funciones correspondientes ya que se han definido funciones para cada una de las opciones principales.

En la Figura 37 se muestra parte del código que permite seleccionar una de las herramientas mostradas en la Figura 36. En el código se comprueba la posición del cursor en el software y se determina en donde hace click en función de posiciones previamente definidas.

```
//TOOLBAR
if (parameter_window==false) {
  for (int i=0; i<11; i++) {
    if (num_option==0) {
      if ((mouseX>rect_icon[(2*i)])&&(mouseX<rect_icon[(2*i)+1])&&(mouseY>22)&&(mouseY<48)) {

        if ((i==4)&&(execution==true)) {
        } else if ((i==5)&&(compiled==false)||(i==5)&&(connected==false)) {
        } else if ((i==6)&&(execution==false)) {
        } else if ((i==7)&&(connected==false)) {
        } else if ((i==9)&&(connected==false)) {
        } else {
          switch(i) {

          case 0:
            FILE(file[0]);
            break;

          case 1:
            FILE(file[1]);
            break;

          case 2:
            FILE(file[2]);
            break;

          case 3:
            FILE(file[4]);
            break;
```

Figura 37: Barra de Herramientas Funcionamiento

Básicamente el código comprueba si se ha hecho un click dentro de alguna zona determinada de acuerdo a las dimensiones de las opciones en pixeles. Si se comprueba que el click ocurre dentro de los límites entonces verifica a quien le corresponde e invoca a la función correspondiente.

3.3.5. State Menu (Menú de Estados)

El menú de estados se compone de 5 íconos de los cuales 4 representan estados del proyecto y 1 de ellos muestra la versión del software (Full o Demo).

Los estados que se pueden mostrar son:

- **Estado de ejecución:** En este estado se muestra si el proyecto se encuentra en ejecución mediante dos íconos representativos y un pequeño texto descriptivo tal como se observa en la Figura 38.

Figura 38: Estado de ejecución

- **Estado modo de edición:** Este estado solo muestra si el proyecto se encuentra habilitado para su edición. Si se encuentra habilitado se indica mediante un ícono de color de lo contrario se mostrará de un color plomo (inactivo). Se observa este estado en la Figura 39.

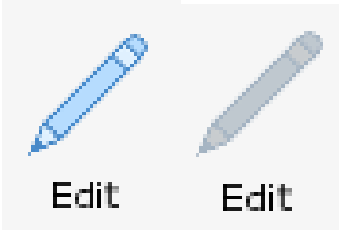


Figura 39: Estado modo de edición

- **Estado de compilación:** Mediante un ícono se indica al usuario si el proyecto ha sido compilado con éxito. La compilación exitosa se indica mediante un ícono de color. El estado de compilación se muestra en la Figura 40.

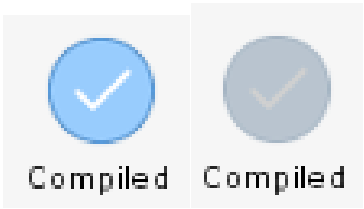


Figura 40: Estado de compilación

- **Estado de conexión:** El estado de conexión indica si el hardware ha logrado establecer una comunicación exitosa con el software. Al igual que en el resto de estados, la conexión se indica mediante un ícono de color y en caso contrario se mostrará el mismo ícono, pero de color plomo tal como se observa en la Figura 41.

Figura 41: Estado de conexión

- **Estado de la licencia:** Finalmente el software indicará si se encuentra una clave de licencia para habilitar todas las herramientas del programa. Si se encuentra una clave el estado que se muestra es *Full y* en caso contrario el estado que se muestra es *Trial* (versión de prueba). Los íconos que muestran este estado se observan en la Figura 42.

Figura 42: Estado de licencia

Los estados cambian en función de las siguientes condiciones:

- **Running** se muestra si **Ejecución** es **Verdadero**
- **Edit mode** se activa si **Ejecución** es **Falso**
- **Not compiled** se muestra si **Compilación** es **Falso**
- **Compiled** se muestra si **Compilación** es **Verdadero**
- **Connected** se muestra si **Conexión** es **Verdadero**
- **Demo** se muestra si **Demo** es **Verdadero**

Las condiciones en mención se implementan en AuroraLD Studio en una función denominada STATUS. Parte de la función implementada se muestra en la Figura 43. En el código tenemos las condiciones IF antes especificadas, así como las imágenes representativas de cada uno de los estados y su respectivo color en función de las condiciones de activación.

```
void STATUS(){

  textSize(9.4);
  if(execution==true){
  noTint();
  image(running_mode,30,60,35,35);
  tint(150,100);
  image(edit_mode,80,60,35,35);
  noTint();
  fill(status_text);
  text("Running",48,105);
  }
  else if(execution==false){
  noTint();
  image(not_running_mode,30,60,35,35);
  image(edit_mode,80,60,35,35);
  fill(status_text);
  text("Stopped",46,105);
  }
  fill(status_text);
  text("Edit",97,105);
```

Figura 43: Menú de estados funcionamiento

Para colocar las imágenes en el software se emplean las funciones previamente mencionadas.

- ***PImage**:* permite declarar variables de tipo imagen.
- ***loadImage**:* permite cargar las imágenes desde el folder *data*.

La declaración de las variables imagen que se utilizan para la implementación del menú de estados se muestra en la Figura 44 y la carga de las imágenes desde el folder data se realiza con el código presentado en la Figura 45.

```
//STATUS ICONS
PImage running_mode;
PImage not_running_mode;
PImage edit_mode;
PImage compile_mode;
PImage connection_status;
PImage demo_mode;
PImage full_mode;
```

Figura 44: Declaración de íconos de estado

```
//STATUS
running_mode = loadImage("RUNNING.png");
not_running_mode = loadImage("NOT_RUNNING.png");
edit_mode = loadImage("EDIT.png");
compile_mode = loadImage("OK.png");
connection_status = loadImage("CONNECTED.png");
demo_mode = loadImage("TRIAL.png");
full_mode = loadImage("FULL.png");
```

Figura 45: Carga de íconos de estado

El menú de estados implementado en AuroraLD Studio se muestra en la Figura 46. Como se detalló antes este menú ayuda a visualizar el estado en el que se encuentra el software AuroraLD Studio.

Figura 46: Menú de estados

3.3.6. Instruction Menu (Menú de Instrucciones)

Al costado derecho del menú de estados se encuentra el menú de instrucciones (ver Figura 16) el cual contiene todas las instrucciones para crear un diagrama Ladder. No es propósito de esta actividad terminar de implementar las instrucciones, sin embargo, se define el espacio, colores, botones, pestañas, etc. De acuerdo con lo definido en el planteamiento, AuroraLD Studio cuenta con 8 categorías de instrucciones diferentes:

- Instrucciones Comunes
- Temporizadores/Contadores
- Comparadores
- Funciones Matemáticas
- Función Move
- Funciones Trigonométricas
- Funciones de Conversión
- Instrucciones Especiales

Las categorías en mención estarán representadas en pestañas de manera que se pueda elegir y encontrar las instrucciones deseadas con mayor facilidad.

Se presenta en la Figura 47 el diseño implementado para el menú de instrucciones contemplando las categorías mencionadas en forma de pestañas. El usuario deberá cambiar entre pestañas para cambiar la categoría y por ende encontrar de manera más rápida las instrucciones que desee emplear en el proyecto.

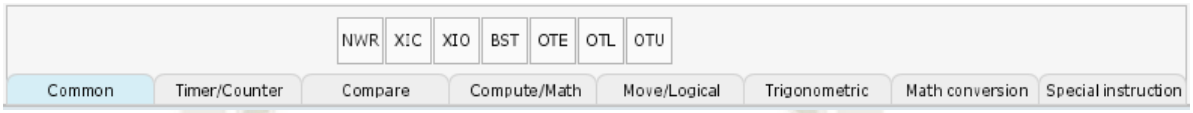


Figura 47: Menú de instrucciones

El menú de instrucciones ha sido dibujado en base a código en Processing sin emplear ningún ícono adicional. Las instrucciones se muestran en pequeños cuadrados con una etiqueta correspondiente a cada instrucción. Las instrucciones y sus nombres están basados en el software RSLogix 5000 de Rockwell Automation.

3.3.7. Left Bar (Barra lateral izquierda)

La barra lateral izquierda (de acuerdo a la Figura 16) brindará opciones adicionales al usuario. Las opciones que se contemplan en esta barra son respecto al proyecto y al programa. La implementación de la barra lateral izquierda consta de 9 íconos que se han definido en función de lo que realizan tal como se muestra en la Figura 48.

Las opciones referentes al proyecto ya existen en la barra de herramientas, pero se vuelven a presentar aquí bajo otros íconos con el propósito de mostrar al usuario como es que se estructura un proyecto realizado en AuroraLD Studio.

Por otro lado, las opciones referentes al software brindan 3 opciones de las cuales 1 ya es conocida (Propiedades del controlador) y las otras dan la posibilidad de modificar el esquema de colores y especificar una licencia para el software. El esquema de colores hará que sea más personalizado y a gusto del usuario.

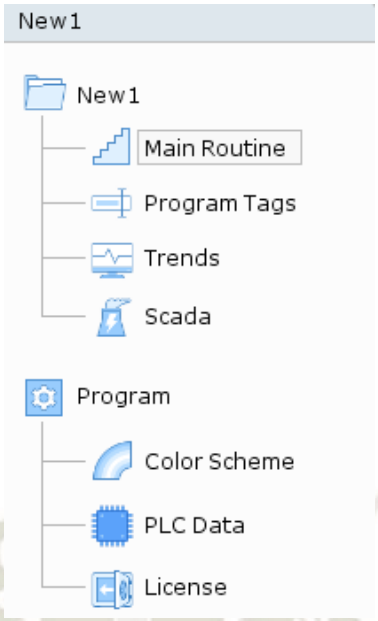


Figura 48: Barra lateral izquierda

3.3.8. Ladder Workspace (Entorno de Trabajo Ladder)

Las distintas instrucciones con las que cuenta AuroraLD Studio son arrastradas a este espacio de trabajo permitiendo así implementar la lógica necesaria para un determinado proyecto. Es en este espacio de trabajo (Figura 49) donde se crea el diagrama Ladder.

Todos los gráficos necesarios para cada una de las instrucciones serán graficados en Processing.

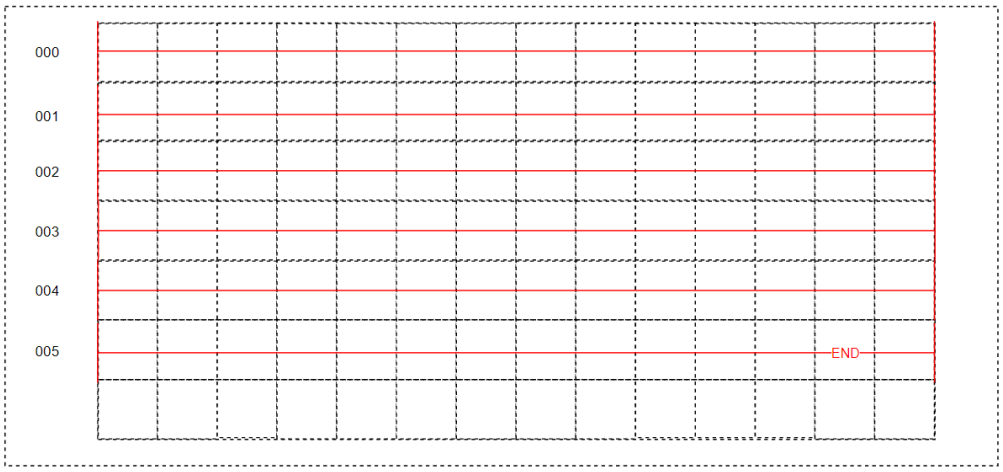


Figura 49: Diseño Preliminar Ladder Workspace

En la Figura 49 se puede observar un diseño preliminar del entorno Ladder, el cual contiene dos rieles (izquierdo y derecho), además las diferentes instrucciones se graficarán en cada espacio teniendo así proporciones conocidas para verificar sus conexiones cuando se requiera ejecutar el programa.

El entorno Ladder está diseñado en base a una malla (Grid) que consta de 14 columnas: 10 empleadas para las entradas (Saving), 3 para las salidas (Computing) y 1 columna que sirve de espacio entre las entradas y salidas tal como se observa en la Figura 49. Adicionalmente cada una de las líneas (Rungs) estarán enumeradas.

En el espacio de trabajo Ladder se ha definido solamente la malla donde irán insertados las instrucciones del diagrama Ladder, así como un Scroll bar para desplazarse por este espacio de trabajo de forma vertical conforme se agreguen más instrucciones. El espacio de trabajo implementado se muestra en la Figura 50.

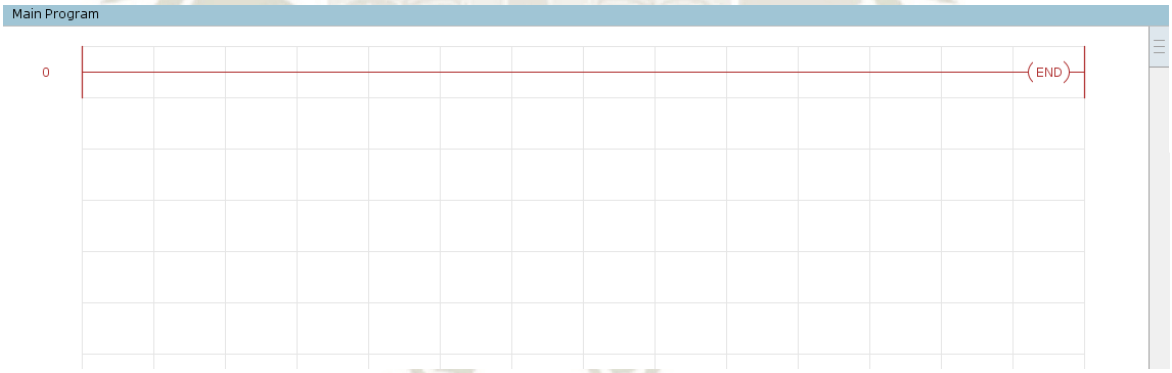


Figura 50: Ladder Workspace (Diseño Preliminar)

La malla guía se implementa en base a rectángulos colocados en un bucle de manera que solo es necesario crear un solo rectángulo para definir toda la malla. El bucle en mención se muestra en la Figura 51. Mediante las iteraciones presentadas se define toda la malla guía.

```
//GRID
fill(255);
stroke(0);
rectMode(CORNERS);
int square_size=70;
int start_grid_x=255;
int sq_size_y=50;
int start_grid_y=160;

for(int j=0;j<108;j++){
for(int i=0;i<15;i++){
rect(start_grid_x+square_size*i,
}
}
```

Figura 51: Parte del código malla guía

El Scroll bar funciona con otro evento mouse denominado mouseWheel, este evento permite detectar cambios en la rueda del mouse de forma que podemos emplear variables para desplazarnos de forma vertical por el entorno en función del avance de la rueda. El código empleado para realizar el desplazamiento vertical empleando la rueda del mouse se muestra en la Figura 52.

```
void mouseWheel(MouseEvent event){
float e=event.getCount();

if(scroll_activate==1){
scroll_pos=scroll_pos+int(e);
scroll_pos=constrain(scroll_pos,160,y_screen-45);
y_scroll=scroll_pos-160;
scroll=y_scroll*scroll_speed;
}
}
```

Figura 52: Evento mouseWheel Ladder Workspace

Cuando las instrucciones sean agregadas y se tenga definido las dimensiones de cada instrucción ya no será necesario contar con la malla guía por lo que comentar el código que la genera es lo más conveniente.

3.3.9. Status Bar (Barra de Estado)

Finalmente se tiene la barra de estado la cuál trabaja de manera similar al menú de estados solo que en vez de utilizar íconos se representan los estados de forma textual. Esta barra estará ubicada en la parte inferior del software. Adicionalmente a los estados esta barra también mostrará errores que se generen en el software al momento de su compilación. En la Figura 53 se observa la barra de estados indicado el modo de edición.

Figura 53: Barra de estados

La barra de estados a implementar solo mostrará los siguientes mensajes:

- **Errors found – Program not compiled** (Cuando se intenta compilar un programa que contiene errores)
- **Program compiled** (Cuando no existen errores y se compila con éxito)
- **Edit mode** (Cuando el programa no se encuentra en ejecución)
- **Running** (Cuando el programa ha sido ejecutado y está corriendo)

```
if (error_counter>0) {
  text("ERRORS FOUND - PROGRAM NOT COMPILED", 5, y_screen-12);
} else if ((compiled==true)&&(execution==false)) {
  text("PROGRAM COMPILED", 5, y_screen-12);
} else if ((compiled==false)&&(execution==false)) {
  text("EDIT MODE", 5, y_screen-12);
} else if (execution==true) {
  text("RUNNING", 5, y_screen-12);
}
```

Figura 54: Código barra de estados

En la Figura 54 se muestra el código necesario para la implementación de la barra de estados. En el código se puede apreciar los diferentes mensajes que la barra de estados es capaz de mostrar al usuario.

Tras haber implementado en el software las distintas opciones ya se tiene una interfaz con todas las herramientas distribuidas. Con esta parte se finaliza la implementación de la interfaz de usuario respecto al entorno Ladder. La interfaz gráfica de usuario de AuroraLD Studio para el entorno mencionado se muestra en la Figura 55.

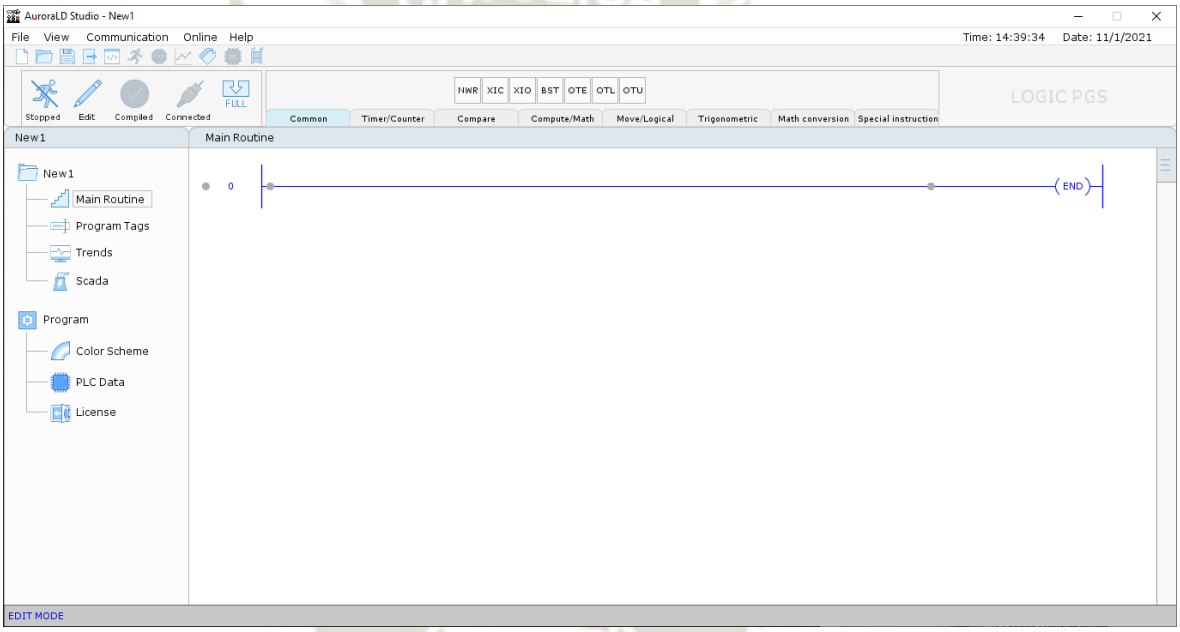


Figura 55: Entorno de AuroraLD Studio

3.4. Definir e implementar las instrucciones Ladder

De acuerdo con lo planificado las instrucciones que se implementarán se presentan en la Tabla 6.

Tabla 6: Instrucciones a implementar

Instrucciones	Descripción
Comunes	Rungs, Contactos NA/NC (XIC, XIO), Branch (BST), Bobinas (OTE), Bobinas Latch (OTL) y Unlatch (OTU).
Temporizadores/Contadores	Temporizador TON, TOF, RTO, Contador ascendente y descendente (CTU, CTD), Bobina de RESET (RES).
Comparadores	Límites (LIM), Igual a (EQU), No igual a (NEQ), Menor que (LES), Mayor que (GRT), Menor o igual que (LEQ) y mayor o igual que (GEQ).
Funciones Matemáticas	Suma (ADD), Resta (SUB), Multiplicación (MUL), División (DIV), Raíz Cuadrada (SQR), Negación (NEG), Módulo (MOD), Valor Absoluto (ABS), Compute (CPT) y Escalamiento (SCP).
Función Mover (asignación)	Mover (MOV)
Funciones Trigonométricas	Arco coseno (ACS), Arcoseno (ASN), Arco tangente (ATN), Coseno (COS), Seno (SIN) y Tangente (TAN)
Funciones de conversión	Radianes (RAD) y Degrees (DEG)
Instrucciones especiales	Control PID discreto (PID)

Las instrucciones en mención se consideran las más utilizadas al momento de crear un proyecto empleando diagramas Ladder. En los siguientes puntos se detalla las variables, funcionamiento, diseño e implementación de cada una de las instrucciones especificadas.

Todas las instrucciones en AuroraLD Studio están basadas en clases y por lo general estas clases contienen las siguientes partes:

- **Variables de la clase:** Definen las variables que emplearán los objetos. También denominados atributos.
- **Constructor:** Es la función principal que permite la creación de un objeto con los atributos propios de una clase.
- **Función *create():*** Muestra el objeto creado.
- **Función *calc():*** Permite realizar el cálculo de una determinada instrucción, es decir, el funcionamiento de una instrucción (sumar, restar, etc.)
- **Función *data():*** Permite pasar la información entre lo calculado en las instrucciones y los tags del programa. También es posible tomar datos de los tags del programa para emplearse en algunas instrucciones que lo requieran. Comunicación bidireccional con la tabla de datos del programa (Program Tags).

A continuación, se detalla más información de cada una de las instrucciones Ladder que se implementarán en AuroraLD Studio.

3.4.1. Instrucciones Comunes (Common)

Dentro de las instrucciones comunes podemos encontrar hasta 7 diferentes instrucciones las cuales se muestran en la Figura 56. Se accede a estas instrucciones al presionar en la pestaña Common.

Figura 56: Instrucciones comunes

Las instrucciones XIC, XIO, OTE, OTL y OTU poseen los siguientes atributos declarados dentro de cada una de las clases respectivas:

- **name:** Hace referencia al tag que está asociado a la instrucción. El tipo de variable es un *String.*
- **type:** El tipo de instrucción (XIC, OTE, etc). El tipo de variable es un *String.*
- **num:** Número de objeto creado. El tipo de variable es un *int.*
- **row_tag:** Fila donde se encuentra el tag asociado a la instrucción. El tipo de variable es un *int.*
- **x:** Determina la posición de la instrucción con respecto a una coordenada x en pixeles. El tipo de variable es un *int.*
- **y:** Determina la posición de la instrucción con respecto a una coordenada y en pixeles. El tipo de variable es un *float.*
- **value:** Al tratarse de una variable booleana el parámetro value solo puede tomar valores entre 0 y 1. El tipo de variable es un *int.*
- **state:** State es una variable booleana que permite conocer el estado actual de la instrucción en función de su valor (0 o 1). El estado puede ser falso o verdadero declarando este parámetro como una variable *boolean.*
- **belong:** Indica la pertenencia de la instrucción a un determinado Rung. El tipo de variable es *int.*

El constructor de las instrucciones mencionadas solo emplea 8 atributos de los mostrados con anterioridad puesto que el parámetro row_tag depende del parámetro name. Conociendo name es posible determinar la fila en donde se encuentra dicho tag (etiqueta). La declaración del constructor de la clase NO se muestra en la Figura 57 en donde se especifica que tipo de variable es cada uno de los atributos previamente declarados.

A partir del constructor se crean los objetos XIC con los atributos mostrados en la Figura 57. Este código es similar en las otras clases de las cuales derivan los objetos XIC, XIO, OTE, OTL y OTU.

```
NO(String name_, String type_, int num_, int x_, float y_, int  value_,
  boolean state_, int belong_) {
  name = name_;
  type=type_;
  num=num_;
  x = x_;
  y = y_;
  value=value_;
  state = state_;
  belong=belong_;
}
```

Figura 57: Constructor de la clase NO

Adicionalmente, es importante mencionar que en AuroraLD Studio existen direcciones para cada una de las etiquetas que se van a tener en el programa: dirección externa (hardware) o una dirección interna (solo en software). Solamente las direcciones externas se podrán ver físicamente en el PLC. Las variables internas funcionan como memorias en el software. Las direcciones externas de las etiquetas para variables digitales (Bool) se dividen en dos: entradas digitales y salidas digitales.

Las direcciones se especifican con dos letras indicando si se trata de una entrada o una salida, un número para indicar el número de ranura y finalmente un número separado por slash (/) indicando el bit correspondiente.

En el caso de las entradas estas van desde DI:1/0 hasta DI:1/5. Por otro lado, las salidas van desde DO:2/0 hasta DO:2/5. Existen dos tags adicionales que no se observan a simple vista, pero están relacionados con los temporizadores y contadores siendo estas

etiquetas EN (Enable) y DN (Done Bit). Estos tags también son parte de las variables booleanas pudiendo ser empleadas por las instrucciones XIC.

Para facilitar la interacción del usuario con estas instrucciones se crea una ventana de parámetros de forma que se puede escoger el tag que se desea emplear y mostrar información adicional relacionada con el tag escogido tal como se muestra en la Figura 58. Toda la información de la ventana de parámetros es obtenida de la tabla de datos del programa.

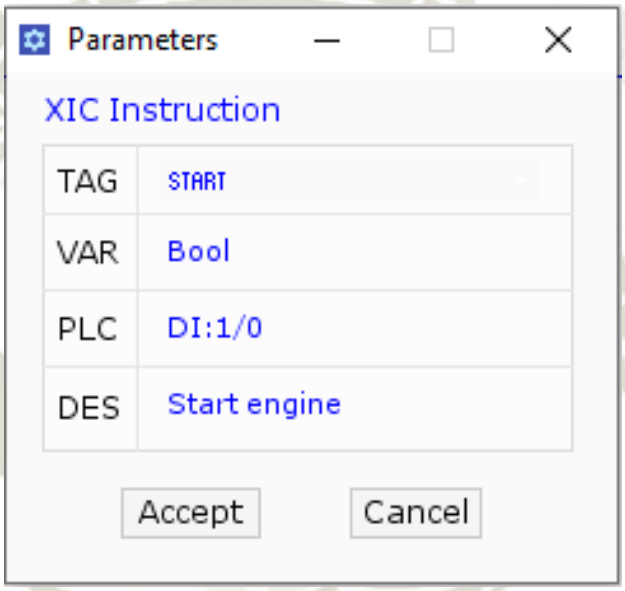


Figura 58: Ejemplo parámetros XIC

La función create() en el interior de las clases implementadas permite mostrar las instrucciones agregadas al Ladder workspace.

3.4.1.1. Nueva Línea (NWR)

Una línea (Rung) representa la instrucción más básica en un diagrama Ladder, básicamente representa el flujo de energía en una determinada línea al conectar el carril de alimentación izquierdo con el derecho. Es sobre esta línea que se colocan las demás

instrucciones creando así las condiciones necesarias para la ejecución de una determinada línea. Esta instrucción se implementa en AuroraLD Studio tal y como se observa en la Figura 59.

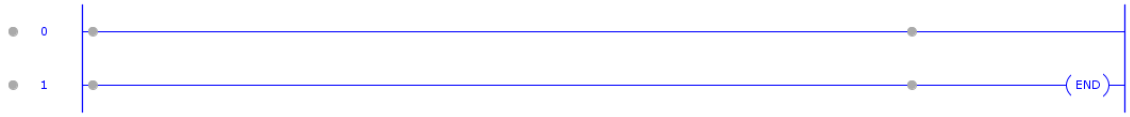


Figura 59: Instrucción nueva línea

En AuroraLD Studio al igual que en otros softwares de programación Ladder siempre existirá una línea en la cual no se permite colocar instrucciones, pero representa el fin del programa. Esta línea en AuroraLD Studio se reconoce con facilidad ya que siempre lleva una única instrucción indicando END o fin del programa.

La ejecución de las líneas (Rungs) en el software siempre se efectúan de arriba hacia abajo y se repite la ejecución una vez que se alcanza la línea que representa el fin del programa (END) hasta que se detenga la ejecución o se desconecte el hardware.

Asimismo, los Rungs están divididos en dos partes: una parte izquierda denominada Saving y una parte derecha denominada Computing tal como se observa en la Figura 60. Saving es también conocido como las entradas y representa las condiciones que se deben de satisfacer para que las salidas (Computing) se activen.

En el Ladder Workspace implementado la parte correspondiente a Saving (condiciones) está representado por las 10 primeras columnas, se deja un espacio en blanco y las tres columnas restantes sirven para la parte de Computing (salidas). Observamos de forma gráfica lo mencionado en la Figura 60.

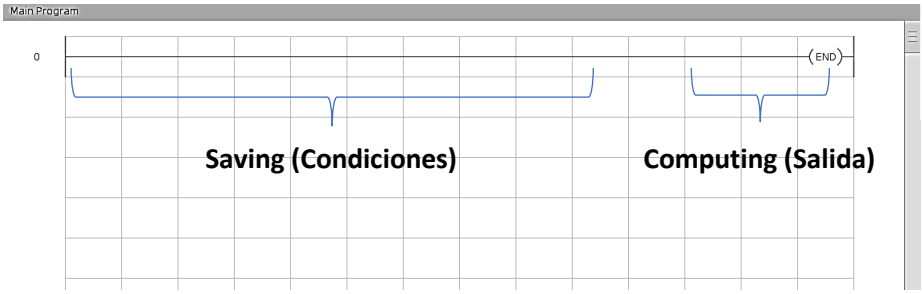


Figura 60: Espacios para instrucciones de entrada y salida

El proyecto está enfocado en el uso de la programación orientada a objetos (POO), por esta razón la implementación de la instrucción Rung está basada en una clase denominada NWR y los objetos (instrucciones que se van agregando al proyecto creados en base a esta clase) se denominan RUNG.

Todos los objetos RUNG han sido declarados como un array de objetos de manera que todos lleven un nombre en común y solamente se diferencie por el índice del array. Ejemplo, RUNG [0], RUNG [50], etc. El valor máximo definido para un array de objetos es 199 permitiendo así crear como máximo 200 Rungs. En la Figura 61 se declaran los objetos RUNG limitando también a su vez la cantidad máxima de objetos que se pueden crear a partir de la clase NWR.

```
//RUNG
NWR RUNG[] = new NWR[200];
int rung_counter=0;
```

Figura 61: Declaración de objeto RUNG

Notamos que en la Figura 61 aparte de la declaración de objetos se necesita también una variable llamada contador de líneas (rung_counter) la cual llevará la cuenta de todas los Rungs que se han creado en un proyecto.

Los atributos característicos de la clase NWR son:

- ***num:*** Representa el número del objeto creado
- **y:** Representa la posición en el eje "y" de los Rungs

El resto de variables mostradas en la Figura 62 permiten la interacción con el resto de instrucciones de manera que se puede armar un diagrama Ladder. Estas variables verifican la existencia de instrucciones sobre una determinada línea.

```
class NWR {
  int num;
  float y;
  boolean draw;
  boolean draw2;
  boolean draw3;
  boolean exist=false;
  boolean exist2=false;
  boolean exist3=false;
```

Figura 62: Clase NWR variables

El constructor de la clase NWR solo emplea los parámetros o atributos característicos tal como se muestra en la Figura 63.

```
NWR(int num_, float y_) {
  num=num_;
  y=y_;
}
```

Figura 63: Constructor de clase NWR

El dibujo y la numeración respectiva de los Rungs se realiza mediante la función create(), el cuál es un método implementando dentro de la clase NWR.

Básicamente el dibujo está compuesto de una línea horizontal y texto al lado izquierdo de la línea indicando así el número respectivo. En la Figura 64 se muestra el código implementado dentro del método create() y que ejecuta lo antes mencionado.

```
//DRAWING
stroke(lines);
strokeWeight(1);
textFont(font11);
fill(lines);
line(300, y-scroll, 1280, y-scroll);
text(num, 265, y-scroll);
```

Figura 64: Función create() NWR

Esta instrucción es agregada al Ladder Workspace arrastrando la instrucción desde el menú de instrucciones y colocándola en donde corresponde dentro del entorno de desarrollo del proyecto. Pese a contar con un límite definido para la creación del objeto RUNG de 200 existe un límite dado por el espacio de trabajo presente en el entorno Ladder siendo así el número máximo de líneas simples que se pueden tener en AuroraLD Studio es 108 líneas, entiéndase por líneas simples todas aquellas instrucciones que se encuentran en la pestaña de “Common”.

3.4.1.2. Contacto normalmente abierto (XIC)

Una instrucción XIC representa un contacto normalmente abierto que puede trabajar tanto con entradas como salidas dentro de AuroraLD Studio. El tipo de variable asignado a esta instrucción es del tipo booleano y se conoce en AuroraLD Studio como BOOL.

Al tratarse de una instrucción booleana significa que solo puede trabajar con variables cuyos datos sean 0 o 1, es decir, entradas o salidas digitales. La instrucción implementada en AuroraLD Studio se observa en la Figura 65.

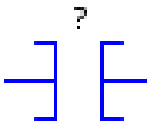


Figura 65: Instrucción XIC

A este tipo de instrucción se le asignará también una etiqueta (tag) que aparecerá sobre la instrucción. Al no tener una etiqueta asignada la etiqueta por defecto es "?" tal como se puede apreciar en la Figura 65.

La instrucción XIC siempre va dibujada sobre un RUNG y nunca ocupa un espacio del lado de Computing. Se pueden poner en el proyecto de distinta forma: en serie o en paralelo considerando que el límite máximo de objetos XIC que se pueden tener es 200. En la Figura 66 se tienen las instrucciones XIC colocadas en el Ladder workspace.

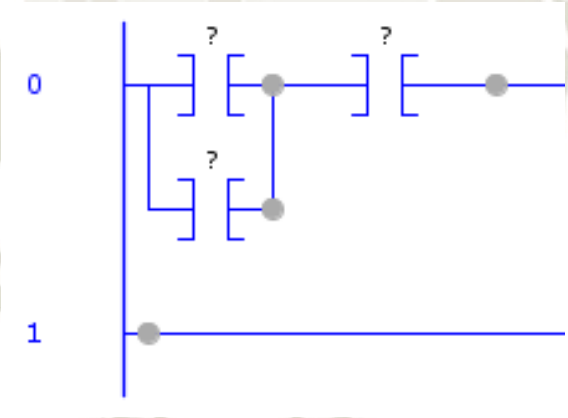


Figura 66: Instrucción XIC en paralelo/serie

En la implementación se ha optado por crear una clase llamada NO y declarar un array de objetos denominados XIC tal cual se muestra en la Figura 67.

```
//XIC
NO XIC[] = new NO[200];
int xic_counter=0;
```

Figura 67: Declaración objeto XIC

3.4.1.3. Contacto normalmente cerrado (XIO)

Una instrucción XIO representa un contacto normalmente cerrado que puede trabajar tanto con entradas como salidas digitales dentro de AuroraLD Studio. El tipo de variable asignado a esta instrucción es del tipo booleano y se conoce en AuroraLD Studio como BOOL.

Al tratarse de una instrucción booleana significa que solo puede trabajar con variables cuyos datos sean 0 o 1, es decir, entradas o salidas digitales. La instrucción XIO implementada en AuroraLD Studio se presenta en la Figura 68.

Figura 68: Instrucción XIO

A este tipo de instrucción se le asignará también una etiqueta (tag) que aparecerá sobre la instrucción. Al no tener una etiqueta asignada la etiqueta por defecto es "?" tal como se puede apreciar en la Figura 68.

La instrucción XIO siempre va dibujada sobre un Rung y nunca ocupa un espacio del lado de Computing. Se pueden poner en el proyecto de distinta forma ya sea en serie o en paralelo considerando que el límite máximo de objetos XIO es 200. En la Figura 69 observamos la instrucción XIO colocada en los Rungs ya sea de forma paralela o serie.

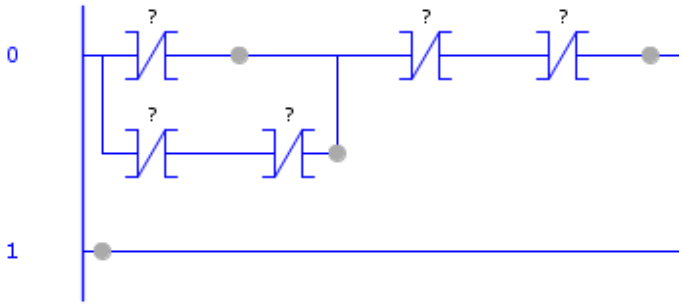


Figura 69: Instrucción XIO paralelo/serie

En la implementación se ha optado por crear una clase llamada NC y declarar un array de objetos denominados XIO tal cual se muestra en la Figura 70.

```
//XIO
NC XIO[] = new NC[200];
int xio_counter=0;
```

Figura 70: Declaración objeto XIO

En la Figura 71 se muestra un ejemplo empleando la etiqueta .DN perteneciente a un temporizador asociado a la instrucción XIO.

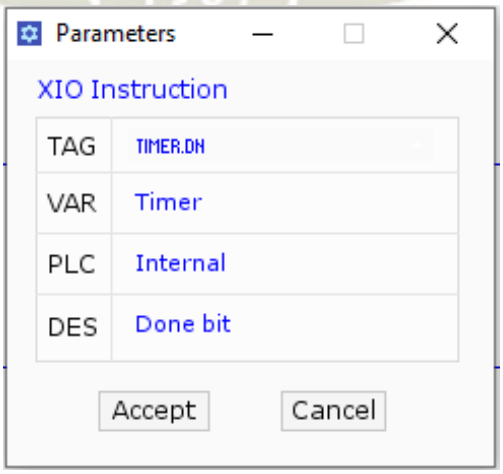


Figura 71: Ejemplo parámetros XIO

3.4.1.4. Branch (BST)

La instrucción Branch (BST) o rama permite la conexión de distintas instrucciones en paralelo. Sin esta instrucción no sería posible colocar instrucciones en paralelo en AuroraLD Studio. La instrucción BST implementada en AuroraLD Studio se muestra en la Figura 72.

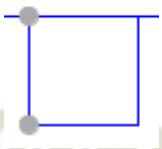


Figura 72: Instrucción BST

Esta es la única instrucción que se puede colocar tanto en la zona de condiciones (entradas) como en la zona de salidas y puede trabajar con todas las instrucciones en AuroraLD Studio. Incluso es posible encadenar varias de estas instrucciones para lograr conectar 2, 3, 4 o más instrucciones en paralelo en caso sea necesario.

No existe un límite en cuanto a la cantidad de instrucciones BST que se pueden colocar en paralelo, pero si existe un límite máximo de objetos BST que se pueden crear, siendo este número 200. Este objeto puede acomodarse de forma automática en función del tamaño de la instrucción que se agregue en su interior. Un ejemplo del uso de la instrucción BST se muestra en la Figura 73.

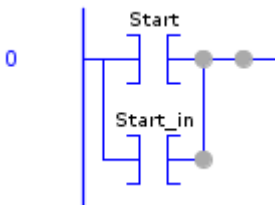


Figura 73: Ejemplo BST dos instrucciones en paralelo

La clase creada para esta instrucción ha sido definida como BRANCH y los objetos se denominan BST. En la Figura 74 se muestra la declaración de objetos respectiva y el contador necesario para saber cuántos objetos de esta clase se van creando en un determinado proyecto.

```
//BST
BRANCH BST[] = new BRANCH[200];
int bst_counter=0;
```

Figura 74: Declaración de objeto BST

La clase BRANCH emplea 7 parámetros principales que se pueden observar en la Figura 75.

```
class BRANCH {
  String type;
  int num;
  int x;
  float y;
  int arm_x;
  int arm_y;
  int belong;
```

Figura 75: Parámetros de clase BRANCH

- **type:** El tipo de esta instrucción es *BST* y el tipo de variable es *String*.
- **num:** La variable num es de tipo *int* y especifica el número del objeto en el momento que fue creado. Ejemplo, 1 si fuera el primero, 2 el segundo, etc.
- **x:** Esta variable es del tipo *int* y hace referencia a la posición de la instrucción BST en el eje X.

- **y:** Esta variable es del tipo *float* y hace referencia a la posición de la instrucción BST en el eje Y.
- **arm_x:** Esta variable hace referencia a la distancia en X que existe entre el extremo derecho de la instrucción BST y su extremo izquierdo. Esta distancia se muestra en la Figura 76.
- **arm_y:** Esta variable hace referencia a la distancia en Y que existe entre el límite superior y el límite inferior de la instrucción BST. Esta distancia se muestra en la Figura 76.

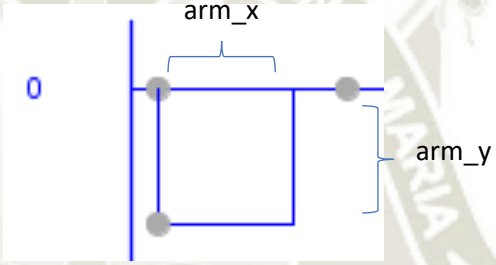


Figura 76: Variables arm_x / arm_y

- **belong:** Finalmente la variable de pertenencia indica con un *int* a que Rung le pertenece la instrucción BST.

Con los parámetros definidos podemos especificar el constructor de la clase BRANCH tal cual se muestra en la Figura 77. En el constructor se ha tomado los atributos más importantes de clase y que ayudan a definir los atributos de los objetos derivados de esta clase.

```
BRANCH(String type_, int num_, int x_, float y_, int arm_x_, int arm_y_, int belong_) {
  type=type_;
  num=num_;
  x=x_;
  y=y_;
  arm_x=arm_x_;
  arm_y=arm_y_;
  belong=belong_;
}
```

Figura 77: Constructor de la clase BRANCH

A diferencia del resto de las instrucciones esta no posee parámetros que sean configurables por el usuario. Por lo tanto, no se le puede asignar una etiqueta y solo sirve como una instrucción para ordenar y crear la lógica necesaria para ejecutar una determinada línea.

3.4.1.5. Bobina (OTE)

Una bobina (OTE) representa la instrucción de salida digital más básica en un diagrama Ladder. Esta instrucción siempre tendrá un valor de salida “0” si las condiciones de ejecución no se satisfacen y un valor de “1” si dichas condiciones se cumplen.

Al igual que las instrucciones XIC y XIO, esta también trabaja con variables booleanas (Verdadero/Falso) y solo admite direccionamiento a variables internas o externas de salida. Las variables externas de salida van desde DO:2/0 hasta DO:2/5. En la Figura 78 se muestra la instrucción OTE implementada en AuroraLD Studio.

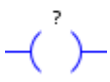


Figura 78: Instrucción OTE

Si no se ha vinculado la instrucción con algún tag se mostrará el tag por defecto sobre la instrucción (“?”). Es importante mencionar que solamente la salida se activa cuando las

condiciones de entrada se cumplen y se desactiva inmediatamente cuando las condiciones de entrada no se satisfacen.

Figura 79: Instrucción OTE activada

En la Figura 79 se observa una instrucción OTE siendo activada por una instrucción de entrada XIC. Cuando se activa el contacto normalmente abierto (XIC) se energiza la bobina que tiene asociada el tag Motor. Las etiquetas .DN y .EN también se pueden emplear con esta instrucción al tratarse de variables internas de salida.

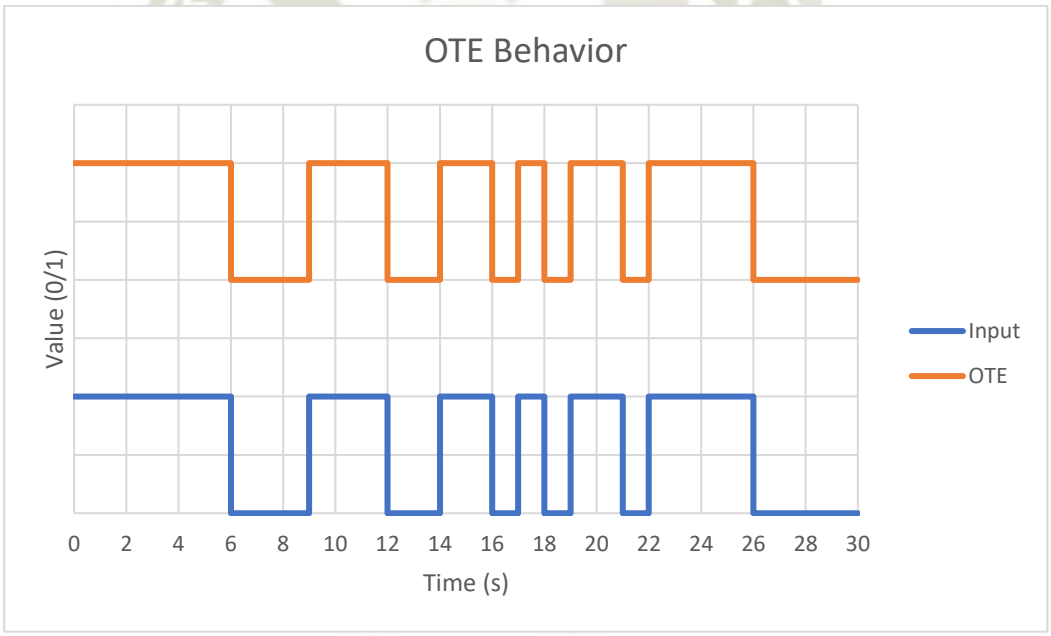


Figura 80: Comportamiento de instrucción OTE

El comportamiento de la instrucción OTE se muestra en la Figura 80. En estas gráficas observamos que ante una entrada (input) de valor alto la instrucción OTE responde de la misma forma. Si el valor de entrada es alto la instrucción OTE también tendrá un valor alto, si por el contrario la entrada es nula o cero la bobina se desactiva pasando al mismo estado que la entrada.

Para implementar la instrucción se ha definido una clase llamada COIL y se ha denominado a los objetos OTE. Como se puede apreciar en la Figura 81 el límite de objetos OTE que se pueden crear es 200. Esta instrucción solo funciona con tags que hayan sido declarados como salidas digitales.

```
//OTE
COIL OTE[] = new COIL[200];
int ote_counter=0;
```

Figura 81: Declaración objeto OTE

3.4.1.6. Bobina Latch (OTL)

Esta instrucción es conocida como Latch (OTL) y básicamente es una instrucción OTE con la ventaja de que almacena su estado. Al satisfacerse las condiciones de entrada, la bobina Latch cambiará de estado pasando de "0" a "1" y se mantendrá en ese estado hasta que la bobina sea reseteada.

Al igual que la instrucción OTE, esta también trabaja con variables booleanas (Verdadero/Falso) y solo admite direccionamiento a variables internas o externas de salida. La instrucción OTL implementada en AuroraLD Studio se muestra en la Figura 82.

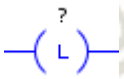


Figura 82: Instrucción OTL

Si no se ha vinculado la instrucción con algún tag la instrucción mostrará la etiqueta por defecto sobre la instrucción ("?"). Es importante mencionar que solamente la salida se activa cuando las condiciones de entrada se cumplen y se desactiva tras un reseteo de la bobina.

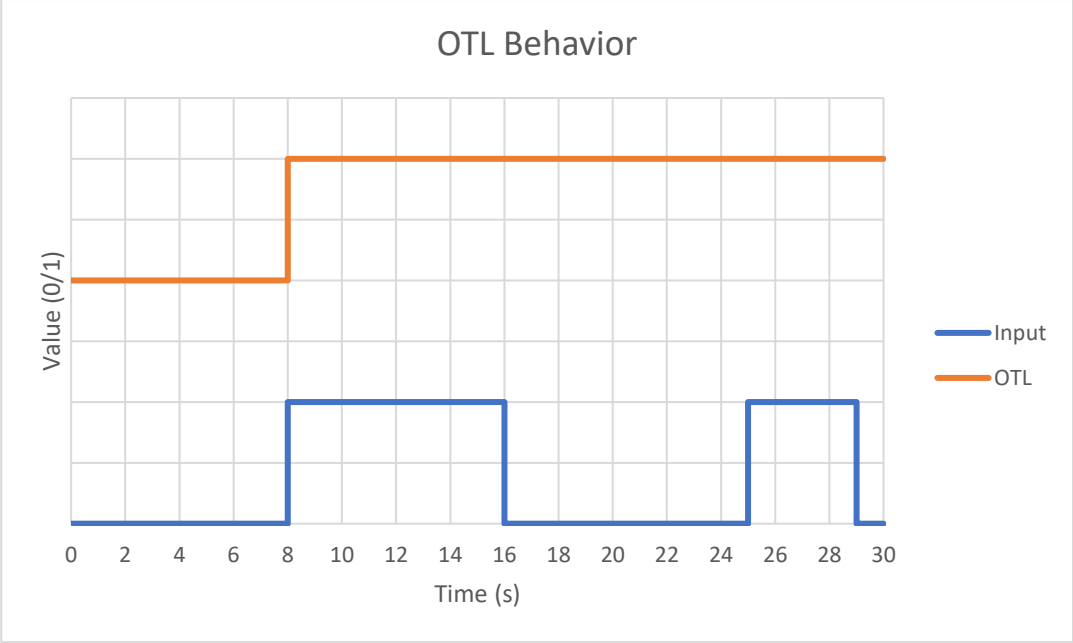


Figura 83: Comportamiento de instrucción OTL

El comportamiento de la instrucción se observa en la Figura 83. Si la entrada pasa de un valor bajo a un valor alto (flanco de subida) la instrucción OTL pasa a un estado alto y se mantiene así sin importar el cambio que se produzca en la entrada.

Para implementar la instrucción se ha definido una clase llamada LATCH y se ha denominado a los objetos OTL. Como se puede apreciar en la Figura 84 el límite de objetos OTL que se pueden crear es 200.

```
//OTL
LATCH OTL[] = new LATCH[200];
int otl_counter=0;
```

Figura 84: Declaración de objeto OTL

3.4.1.7. Bobina Unlatch (OTU)

Una bobina Unlatch permite resetear aquellas bobinas que se han quedado enclavadas (Latch). Cuando se cumplen las condiciones de entrada se activa la instrucción OTU desactivando así todas aquellas bobinas que llevan el mismo tag.

Al igual que la instrucción OTE, esta también trabaja con variables booleanas (Verdadero/Falso) y solo admite direccionamiento a variables internas y externas de salida.

Las variables externas de salida van desde DO:2/0 hasta DO:2/5 (lo que soporta el software por defecto). En la Figura 85 se observa la instrucción OTU implementada en AuroraLD Studio.

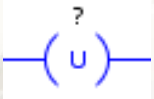


Figura 85: Instrucción OTU

Si no se ha vinculado la instrucción con algún tag, la instrucción mostrará el tag por defecto sobre la instrucción (“?”). El propósito de esta instrucción es resetear las instrucciones OTL.

Para implementar la instrucción se ha definido una clase llamada UNLATCH y se ha denominado a los objetos como OTU. El límite máximo de objetos OTU que se pueden crear es 200. En la Figura 86 se muestra el código que se requiere para declarar los objetos que derivan de la clase UNLATCH.

```
//OTU
UNLATCH OTU[] = new UNLATCH[200];
int otu_counter=0;
```

Figura 86: Declaración de objetos OTU

Es importante mencionar que las instrucciones OTE, OTL y OTU al ser instrucciones de salida solo pueden colocarse en la zona de Computing (salidas). Mientras que las instrucciones XIC y XIO solo pueden colocarse en la zona de Saving (Entradas).

3.4.2. Temporizadores y Contadores

Dentro de las instrucciones de temporizadores y contadores podemos encontrar hasta 6 diferentes instrucciones. Se accede a estas instrucciones al presionar en la pestaña Timer/Counter. Las instrucciones disponibles se muestran en la Figura 87.

Figura 87: Instrucciones de temporizadores y contadores

Las instrucciones Ladder de temporizadores y contadores funcionan con las siguientes variables:

a) Done bit (DN)

Salida digital que indica cuando el acumulador ha llegado al valor de Preset. En numerosos softwares de programación Ladder se conoce a esta variable como ".DN" y por lo general acompaña al tag del temporizador o contador. La variable es también representada de la misma forma en AuroraLD Studio. Por ejemplo: "TEST.DN", donde TEST es el nombre del temporizador y ".DN" es el Done bit. Esta variable tiene un valor de "0" cuando la condición no se satisface (Acumulador no es igual al valor de Preset) y un valor de "1" cuando se satisface la condición (Acumulador es igual al valor de Preset).

b) Enable (EN)

Salida digital que indica que un temporizador o contador está energizado, es decir, activo. En AuroraLD Studio se denomina a la variable como ".EN" y por lo general esta variable acompaña al tag del temporizador o contador. Por ejemplo: "CONT.EN", donde CONT es el tag del contador y ".EN" es la variable Enable. Asume un valor de "1" cuando se activa el temporizador o contador y un valor de "0" cuando se desactiva.

c) Acumulador (ACC)

El acumulador es una salida analógica e indica el valor actual del contador o temporizador. El acumulador puede incrementar o disminuir dependiendo de la instrucción. Esta variable siempre es del tipo entero (int). En AuroraLD Studio y en otros softwares de programación Ladder se conoce a esta variable cómo ".ACC" y suele acompañar al tag del temporizador o contador. Por ejemplo: "TEMP1.ACC", donde TEMP1 hace referencia al nombre del temporizador y ".ACC" al acumulador. Si en algún punto del proyecto se invoca a esta variable siempre se tendrá el valor actual.

d) Preset (PRE)

Esta variable es una entrada analógica y determina el valor que debe ser alcanzado por el temporizador o contador para que se active DN. Este parámetro se puede configurar en la ventana de parámetros de cada instrucción. Se representa por ".PRE" y siempre va acompañado de un tag de temporizador o contador.

Existe también una variable adicional que solo se puede modificar a través de los parámetros de los temporizadores denominado "Time Base". El "Time Base" o tiempo base indica como se realiza la cuenta en el temporizador, es decir, permite elegir entre milisegundos, segundos, minutos, etc. En la Tabla 7 se muestran los tiempos bases que se pueden escoger para los temporizadores.

Tabla 7: Tiempo base

Unit	Tiempo Base (TMB)
Milisegundos (ms)	0.001
Centisegundos (cs)	0.01
Decisegundos (ds)	0.1
Segundos (s)	1
Minutos (min)	60

Las instrucciones de contadores y temporizadores se basan en clases independientes cuyos atributos se muestran en la Figura 88. Los atributos se detallan también a continuación:

- **name:** Es un atributo de tipo String y hace referencia al tag al cual se ha asociado una determinada instrucción.
- **type:** Atributo de tipo String que hace referencia al tipo de instrucción, en este caso TON.
- **num:** Hace referencia al número de objeto creado siendo siempre un valor entero y por ende un atributo tipo int.
- **x:** Coordenada X en pixeles donde se ubica el objeto TON. Este atributo es de tipo int.
- **y:** Coordenada Y en pixeles donde se ubica el objeto TON. Este atributo es de tipo float.
- **state:** Variable de tipo boolean. Indica el estado actual del temporizador si está activado su valor será True, por el contrario, si está desactivado su valor será False.
- **belong:** Esta variable de tipo entero (int). Indica la pertenencia de la instrucción a un Rung.
- **t_base:** Variable de tipo float. Indica el tiempo base escogido en la ventana de parámetros de la instrucción.

- **preset:** Variable de tipo entero (int). Indica el valor de preset al cual se ha configurado la instrucción.
- **acc:** Hace referencia al acumulador de la instrucción. Este atributo es de tipo int.
- **en:** Variable booleana (True/False) que indica si la instrucción está activa o no.
- **dn:** Variable booleana (True/False) que indica si el acumulador de la instrucción es mayor o igual al preset.

```
class TIMER_ON {
  String name;
  String type;
  int num;
  int row_tag = -1;
  int x;
  float y;
  boolean state;
  int belong;
  float t_base;
  int preset;
  int acc;
  boolean en;
  boolean dn;
  int previous_time;
  int actual_time;
```

Figura 88: Atributos clase TIMER_ON

Con los atributos definidos se puede declarar el constructor de la clase, el cual se muestra en la Figura 89. El código mostrado es empleado también por los contadores, pero sin considerar el atributo tiempo base, tiempo previo y tiempo actual.

```
TIMER_ON(String name_, String type_, int num_, int x_, float y_,
  boolean state_, int belong_, float t_base_, int preset_,
  int acc_, boolean en_, boolean dn_) {
  name=name_;
  type=type_;
  num=num_;
  x=x_;
  y=y_;
  state=state_;
  belong=belong_;
  t_base=t_base_;
  preset=preset_;
  acc=acc_;
  en=en_;
  dn=dn_;
}
```

Figura 89: Constructor de clase TON

3.4.2.1. Temporizador On-Delay (TON)

El temporizador TON permite la activar otra instrucción después de haber transcurrido un tiempo determinado. Este temporizador cuenta con las variables: “DN”, “EN”, “ACC” y “PRE”. La instrucción implementada en AuroraLD Studio se muestra en la Figura 90.

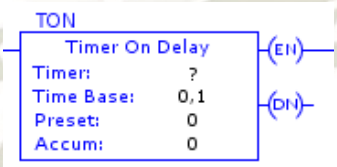


Figura 90: Temporizador TON

Es importante mencionar que esta instrucción se considera una instrucción de salida por lo que solo es posible agregar esta instrucción en el lado derecho del diagrama Ladder. Esta instrucción además se activa por un flanco de subida y mientras se mantenga se realiza la cuenta. Por otro lado, si se quita la señal que activa el temporizador este se desactiva

reseteando el acumulador. En AuroraLD Studio a la instrucción TON solo se le puede asignar tags del tipo Timer cuya dirección sea interna.

Cuando el temporizador es activado EN pasa a "1" y el valor ACC empieza a incrementar. Si el valor del ACC alcanza el valor de PRE entonces DN pasa a "1". Si en cualquier momento el temporizador se desactiva, EN pasa a "0", el valor ACC pasa a "0" y si DN estaba activo también pasa a "0". El comportamiento previamente descrito para el temporizador TON se puede observar en la Figura 91.

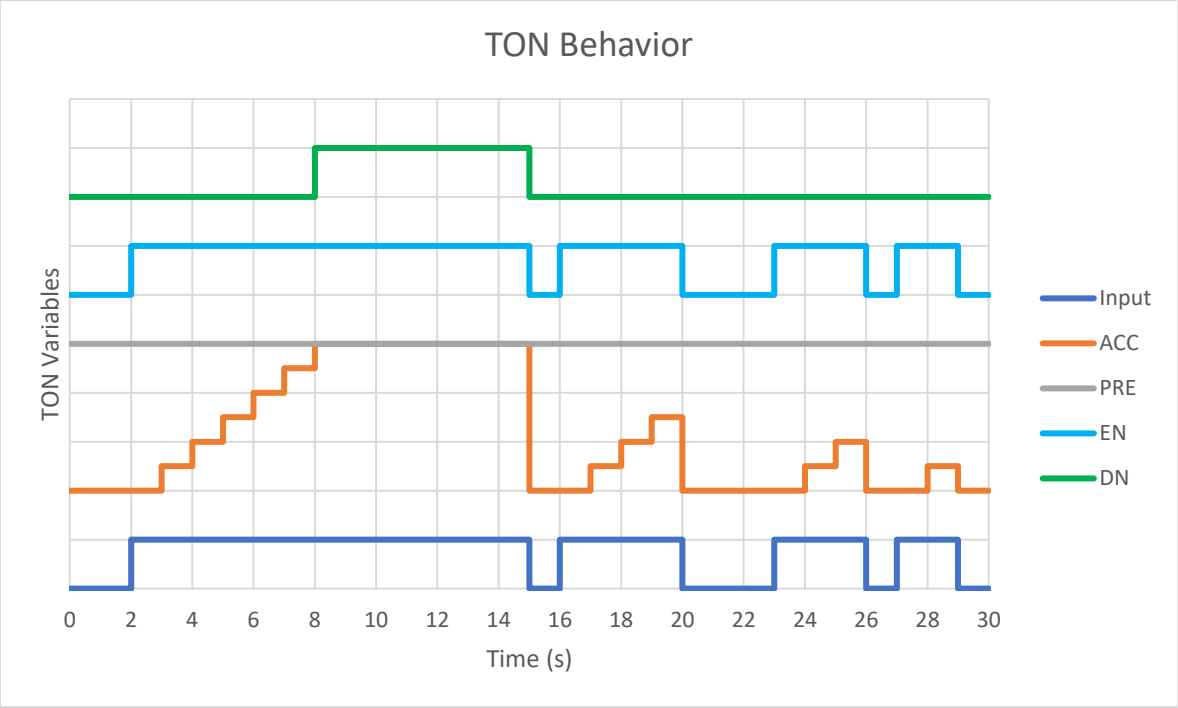


Figura 91: Comportamiento de instrucción TON

Para implementar esta instrucción se crea una clase denominada TIMER_ON. También se declaran los objetos denominados TON y un contador para saber cuántas de estas instrucciones han sido agregadas al proyecto. Cabe destacar que al igual que en las instrucciones anteriores el límite de objetos TON que se pueden crear es 200 tal como se observa en la Figura 92.

```
//TON
TIMER_ON TON[] = new TIMER_ON[200];
int ton_counter=0;
```

Figura 92: Declaración de objeto TON

3.4.2.2. Temporizador Off-Delay (TOF)

A diferencia del temporizador TON, el temporizador TOF se emplea para activar una salida mientras el temporizador está activo. Cuando el Acumulador en el temporizador alcanza el valor de Preset se desactiva el pulso de salida del temporizador (DN). Este temporizador también cuenta con las variables "DN", "EN", "ACC" y "PRE". El temporizador TOF permite modificar el tiempo base configurando la unidad de tiempo en milisegundos, segundos, minutos, etc. La instrucción TOF implementada en AuroraLD Studio se muestra en la Figura 93.

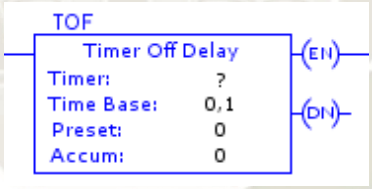


Figura 93: Instrucción TOF

El temporizador TOF en AuroraLD Studio es considerado una instrucción de salida por lo que solo se puede agregar esta instrucción en la zona derecha del Ladder Workspace. El temporizador se activa solo con un flanco de bajada reseteando el acumulador e iniciando su incremento.

La variable "DN" pasa de un valor "0" a "1" cada vez que se cumplen las condiciones de entrada en el Rung y se mantiene en ese estado hasta que el acumulador alcanza el valor de preset.

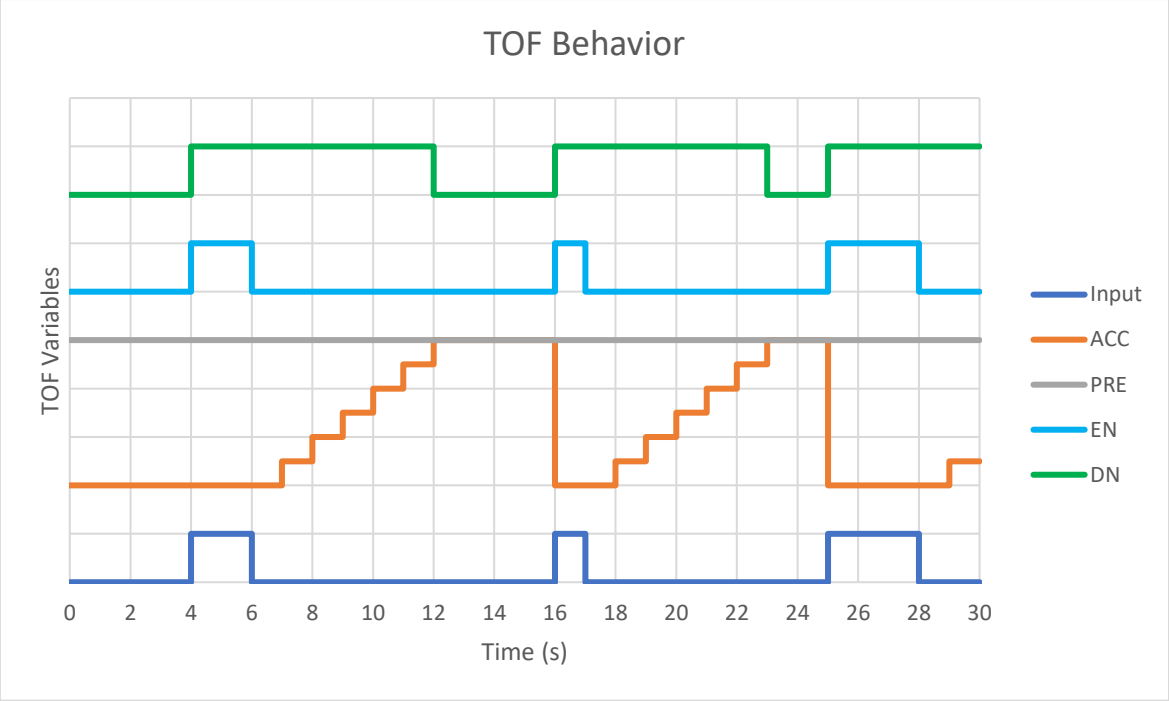


Figura 94: Comportamiento instrucción TOF

En la Figura 94 se observa que al producirse un flanco de bajada en la entrada el temporizador inicia su cuenta hasta alcanzar el valor de preset. Cuando el preset es alcanzado el valor de DN pasa a 0. También observamos que el valor de DN pasa a un estado alto con un flanco de subida.

La implementación de esta instrucción se realiza empleando Processing determinando para ello una clase denominada TIMER_OFF. Esta clase contiene todos los atributos y métodos necesarios de manera que permitan crear objetos. Los objetos de la clase TIMER_OFF se conocen como TOF y el límite máximo de objetos TOF que permite crear AuroraLD Studio es 200 tal como se muestra en la Figura 95.

```
//TOFF
TIMER_OFF TOF[] = new TIMER_OFF[200];
int tof_counter=0;
```

Figura 95: Declaración de objeto TOF

3.4.2.3. Temporizador Retentivo (RTO)

El temporizador retentivo se diferencia del resto de temporizadores al poder almacenar el acumulador así existan flancos de subida o bajada en las condiciones de entrada. Si este temporizador está realizando su cuenta y dejan de cumplirse las condiciones de entrada el valor del acumulador se mantiene y no se resetea como en los otros temporizadores.

La única forma de resetear el valor del acumulador en un temporizador retentivo es a través de una instrucción RES (Reset). Este temporizador se activa con un flanco de subida y solo puede ser asociado a aquellos tags que se han declarado como Timer. Al igual que el resto de temporizadores, este temporizador también emplea las variables "DN", "EN", "ACC" y "PRE". En la Figura 96 se observa la instrucción RTO implementada en AuroraLD Studio.

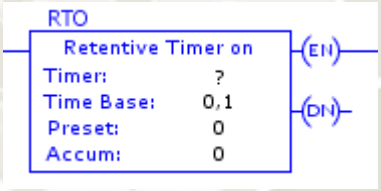


Figura 96: Instrucción RTO

La cuenta solo incrementa una vez que se han cumplido las condiciones de entrada y se detiene si es que dichas condiciones no se cumplen. Esta instrucción se considera como una instrucción de salida en AuroraLD Studio.

El funcionamiento es similar a una instrucción TON. Se activa "DN" una vez que el temporizador ha alcanzado en valor de preset, este comportamiento se observa en la Figura 97. Esta instrucción permite modificar el tiempo base pudiendo así escoger milisegundos, segundos, minutos, etc.

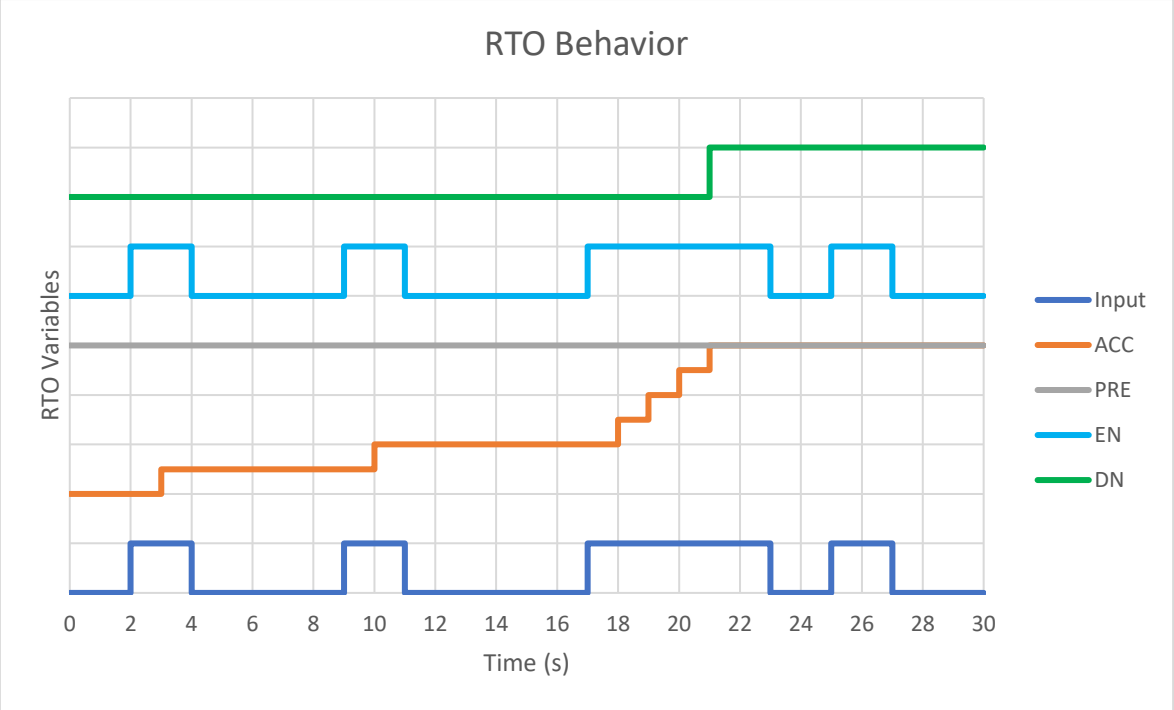


Figura 97: Comportamiento instrucción RTO

La implementación de esta instrucción comienza con una clase denominada TIMER_RTO. En esta clase se han definido todos los parámetros necesarios para la construcción de los objetos RTO. Para poder hacer uso de una clase es necesario definir cuáles serán los objetos que se crearán a partir de la clase en mención y para ello se emplea el código mostrado en la Figura 98.

```
//RTO
TIMER_RTO RTO[] = new TIMER_RTO[200];
int rto_counter=0;
```

Figura 98: Declaración de objeto RTO

3.4.2.4. Contador Ascendente (CTU)

El contador ascendente o CTU es una instrucción que permite llevar la cuenta en una variable denominada acumulador (ACC). Se activa por un flanco de subida en las condiciones de entrada provocando así el incremento del acumulador en una unidad. Esta

instrucción también emplea las variables “DN”, “EN”, “ACC” y “PRE”. A diferencia de los temporizadores, los contadores no requieren especificar un tiempo base por lo que dicho parámetro en la instrucción es inexistente.

Un contador resulta útil para saber cuántas veces se ha llevado a cabo un determinado proceso, cuantas botellas van pasando en una faja, cuantas veces se ha encendido el motor, etc. Esta instrucción solo puede ser asociada a aquellos Tags de tipo Counter. En la Figura 99 se muestra la instrucción CTU implementada en AuroraLD Studio.

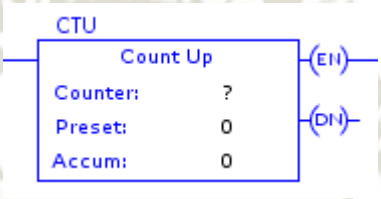


Figura 99: Instrucción CTU

La salida “DN” de este contador se activará siempre y cuando el acumulador sea mayor o igual al valor del preset. Es importante mencionar que en esta instrucción, el acumulador siempre incrementará y su valor solo regresará a 0 siempre y cuando el contador sea reseteado a través de una instrucción RES, comportamiento que se presenta en la Figura 100.

Es común encontrar en diagramas Ladder instrucciones CTU y CTD enlazadas con el mismo tag de manera que se comparta el mismo tag y por ende la misma variable del acumulador permitiendo así crear una cuenta ascendente y descendente.

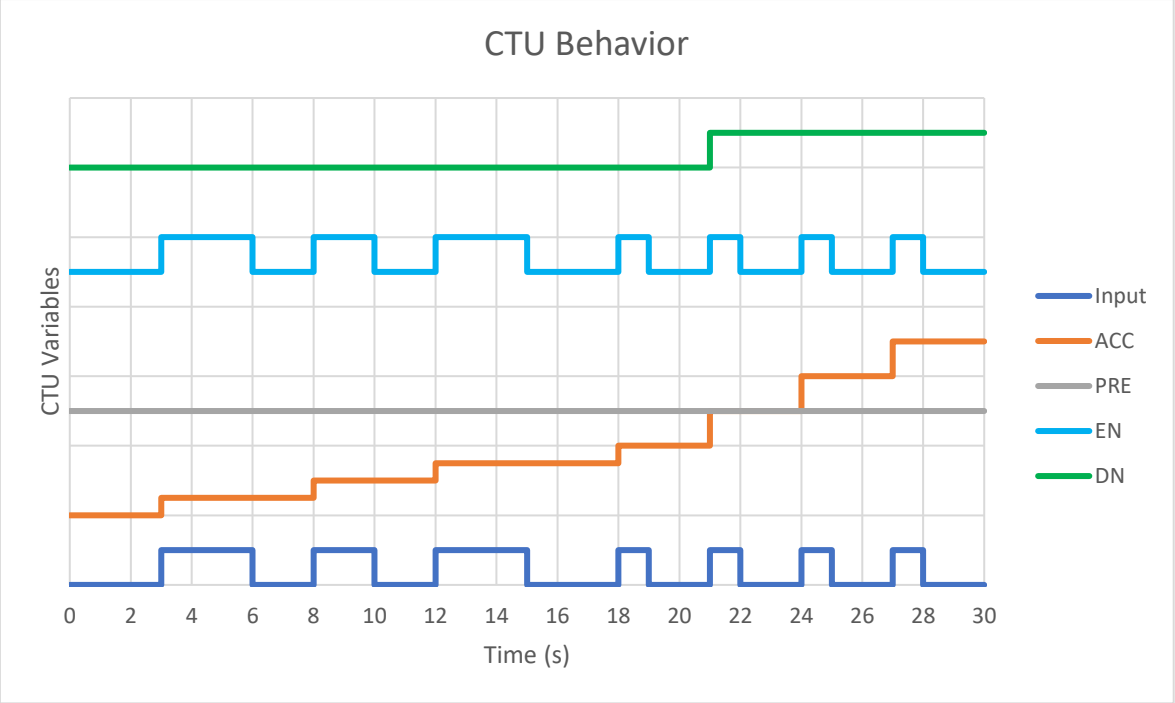


Figura 100: Comportamiento instrucción CTU

Para implementar esta instrucción primero se define una clase en Processing denominada COUNTER_UP. Esta clase permitirá la creación de objetos CTU definiendo un máximo de objetos tal como se muestra en la Figura 101.

```
//CTU
COUNTER_UP CTU[] = new COUNTER_UP[200];
int ctu_counter=0;
```

Figura 101: Declaración de objetos CTU

3.4.2.5. Contador Descendente (CTD)

El contador descendente o CTD es una instrucción que permite llevar la cuenta en una variable denominada acumulador (ACC). Se activa por un flanco de subida en las condiciones de entrada provocando así la disminución del acumulador en una unidad. Esta instrucción también emplea las variables “DN”, “EN”, “ACC” y “PRE”. A diferencia de los

temporizadores, los contadores no requieren especificar un tiempo base por lo que dicho parámetro en la instrucción es inexistente.

Además, es importante mencionar que esta instrucción solo puede estar asociada a Tags del tipo Counter. En la Figura 102 se muestra la instrucción CTD implementada.

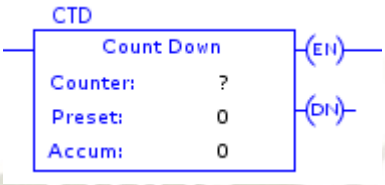


Figura 102: Instrucción CTD

La salida "DN" de este contador se activará siempre y cuando el acumulador sea mayor o igual al valor del preset. Es importante mencionar que en esta instrucción el acumulador siempre disminuirá y su valor solo regresará a 0 siempre y cuando el contador sea reseteado a través de una instrucción RES, el comportamiento previamente descrito se muestra en la Figura 103.

Es común encontrar en diagramas Ladder instrucciones CTU y CTD enlazadas con el mismo tag de manera que se comparte la misma variable del acumulador permitiendo así crear una cuenta ascendente y descendente.

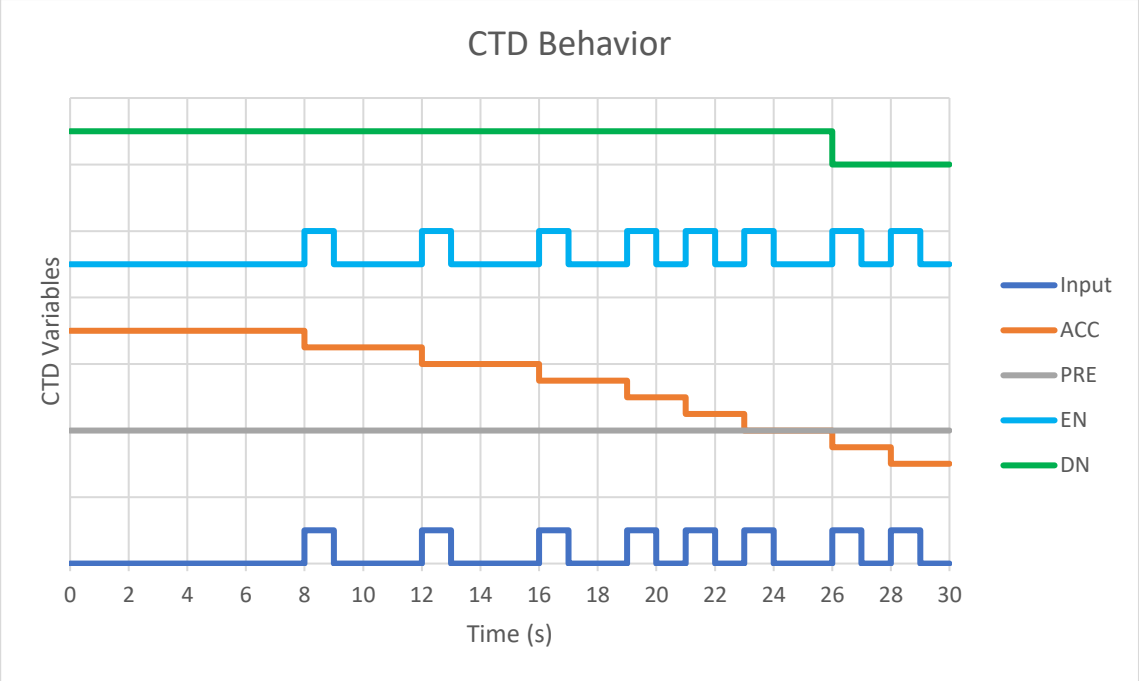


Figura 103: Comportamiento instrucción CTD

La implementación de la instrucción está basada en la clase COUNTER_DOWN y al tratarse de un contador se definen los mismos atributos mostrados en la instrucción CTU.

Para poder emplear una clase y empezar a crear objetos es necesario declarar los objetos y asociarlos con la clase correspondiente tal como se muestra en la Figura 104.

```
//CTD
COUNTER_DOWN CTD[] = new COUNTER_DOWN[200];
int ctd_counter = 0;
```

Figura 104: Declaración objeto CTD

3.4.2.6. Reset (RES)

Esta instrucción permite resetear temporizadores y contadores haciendo que la variable acumuladora (ACC) regrese al valor 0. Generalmente es empleado con las instrucciones RTO, CTU y CTD. Cuando las condiciones de entrada en el Rung se satisfacen la instrucción RES se activa reseteando el acumulador vinculado al Tag al que se ha asociado

la instrucción RES. Cabe destacar que esta instrucción solo puede ser asociada con Tags del tipo Timer/Counter. En la Figura 105 se observa la instrucción RES implementada en AuroraLD Studio.

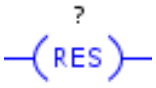


Figura 105: Instrucción RES

El funcionamiento de una instrucción RES es muy parecida al de una bobina y es considerada en AuroraLD Studio como una instrucción de salida digital. La implementación de esta instrucción se realiza empleando la programación orientada a objetos. Se crea una clase denominada RESET y se declaran hasta un máximo de 200 objetos denominados RES tal cual se muestra en la Figura 106.

```
//RES
RESET RES[] = new RESET[200];
int res_counter = 0;
```

Figura 106: Declaración de objetos RES

Los atributos con los que cuenta la clase RESET se muestran en la Figura 107, en esta figura se muestra el nombre de los atributos, así como el tipo de variable dentro del entorno de Processing. A continuación, se detalla cada uno de estos atributos.

```
class RESET {
  String name;
  String type;
  int num;
  int row_tag =-1;
  int x;
  float y;
  boolean state;
  int belong;
```

Figura 107: Atributos principales RESET

- **name:** Atributo de tipo String que hace referencia al tag que tiene asociado.
- **type:** Atributo de tipo String que hace referencia al tipo de instrucción, en este caso RES.
- **num:** Atributo de tipo int que hace referencia al número de objeto creado.
- **x:** Coordenada X en pixeles de la instrucción RES y está definido como un atributo de tipo int.
- **y:** Coordenada Y en pixeles de la instrucción RES y está definido como un atributo de tipo float.
- **state:** Almacena el estado de la instrucción (True/False) dependiendo de las condiciones de entrada en el Rung. Se declara como una variable boolean.
- **belong:** Atributo de tipo int que indica a que Rung pertenece una determinada instrucción RES.

La instrucción RES es tratada en AuroraLD Studio como una instrucción de salida, por lo que se debe arrastrar la instrucción al lado derecho del Ladder Workspace para que pueda ser agregada al proyecto.

3.4.3. Comparadores

Dentro de las instrucciones de comparación podemos encontrar hasta 7 diferentes instrucciones. Se accede a estas instrucciones al presionar en la pestaña Compare. Todas las instrucciones de comparación son consideradas instrucciones de entrada en AuroraLD Studio. Las instrucciones de comparación implementadas se muestran en la Figura 108.

Figura 108: Instrucciones de comparación

Para implementar las instrucciones se emplean atributos son compartidos por el resto de instrucciones de comparación a excepción de la instrucción Límite. Los atributos en mención se muestran a continuación:

- **type:** Atributo de tipo string que hace referencia al tipo de instrucción LES, GRT, GET, etc.
- **num:** Identificador de la instrucción.
- **x:** Posición X en pixeles de la instrucción.
- **y:** Posición Y en pixeles de la instrucción.
- **sourceA_s:** String que hace referencia al tag asociado con el primer valor. Esta variable también puede ser de tipo NUM, es decir, un valor fijo no dependiente de ningún tag declarado en los Program Tags.
- **sourceB_s:** String que hace referencia al tag asociado con el segundo valor. Esta variable también puede ser de tipo NUM, es decir, un valor fijo no dependiente de ningún tag declarado en los Program Tags.
- **sourceA:** Atributo de tipo Float cuyo valor está dado en función del tag asociado (Primer valor).

- **sourceB:** Atributo de tipo Float cuyo valor está dado en función del tag asociado. (Segundo valor).
- **state:** Atributo de tipo Booleano que indica la activación de la instrucción.
- **belong:** Atributo de pertenencia que indica a que línea pertenece una determinada instrucción.

Estos atributos se declaran dentro de cada una de las respectivas clases tal como se muestra en la Figura 110. En esta figura se observan los atributos declarados para la instrucción igualdad (EQU).

```
class EQUAL {
  String type;
  int num;
  int x;
  float y;
  String sourceA_s;
  String sourceB_s;
  float sourceA;
  float sourceB;
  boolean state;
  int belong;
```

Figura 109: Atributos de la clase EQUAL

3.4.3.1. Límite (LIM)

La instrucción Límite es considerada en AuroraLD Studio como una instrucción de entrada la cual se activa siempre y cuando la variable denominada TEST se encuentre entre los límites superior e inferior (High & Low). La instrucción límite implementada en AuroraLD Studio tiene la forma mostrada en la Figura 111.

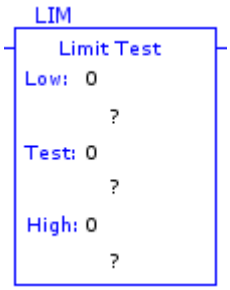


Figura 110: Instrucción LIM

El funcionamiento de esta instrucción corresponde básicamente a la siguiente ecuación matemática:

$$Low \leq Test < High \rightarrow Activates$$

Se crea una clase denominada LIMIT y se declaran como máximo 200 objetos creados a partir de esta clase denominados LIM. En la Figura 111 observamos la declaración de los objetos LIM basados en la clase LIMIT.

```
//LIMIT
LIMIT LIM[] = new LIMIT[200];
int lim_counter;
```

Figura 111: Declaración de objetos LIM

Los atributos con los que cuenta la clase LIMIT son los siguientes:

- **type:** Atributo de tipo string que hace referencia al tipo de instrucción. En este caso type siempre es equivalente a LIM.
- **num:** Hace referencia al número de objeto. Cada vez que se crea una instrucción en AuroraLD Studio se le asocia un número el cual permite identifcar una determinada instrucción.
- **x:** Posición X en pixeles de la instrucción.

- **y:** Posición Y en pixeles de la instrucción.
- **low_s:** String que hace referencia al tag asociado con el límite inferior de la instrucción LIM.
- **high_s:** String que hace referencia al tag asociado con el límite superior de la instrucción LIM.
- **test_s:** String que hace referencia al tag asociado con el valor de prueba, es decir, el valor que se verifica si se encuentre dentro de los límites superior e inferior.
- **low:** Atributo de tipo Float que almacena el valor del límite inferior.
- **high:** Atributo de tipo Float que almacena el valor del límite superior.
- **test:** Atributo de tipo Float que almacena el valor de prueba.
- **state:** Estado de la instrucción. Este atributo indica si la instrucción se encuentra activa o no tras haberse cumplido las condiciones de entrada de su línea correspondiente.
- **belong:** Variable que indica pertenencia a una determinada línea. Este atributo resulta importante para la ejecución del diagrama Ladder.

En la Figura 112 se muestra la declaración de los distintos atributos previamente mencionados, también se definen el tipo de variable al que hace referencia cada uno de los atributos.

```
class LIMIT {
  String type;
  int num;
  int x;
  float y;
  String low_s; //LOW TAG
  String high_s; //HIGH TAG
  String test_s; //TEST TAG
  float low;
  float high;
  float test;
  boolean state;
  int belong;
```

Figura 112: Atributos de la clase LIMIT

3.4.3.2. Igualdad (EQU)

La instrucción Igualdad se activa siempre y cuando exista igualdad entre las variables SourceA y sourceB.

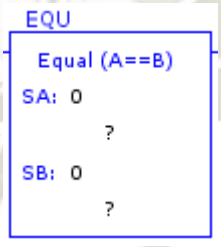


Figura 113: Instrucción EQU

El funcionamiento de esta instrucción corresponde básicamente a la siguiente ecuación:

$$SA = SB \rightarrow Activates$$

A partir de esta clase se crean los objetos denominados EQU y se declara como máximo 200 instrucciones de este tipo tal como se muestra en la Figura 114.

```
//EQUAL
EQUAL EQU[] = new EQUAL[200];
int equ_counter;
```

Figura 114: Declaración de objetos EQU

3.4.3.3. Desigualdad (NEQ)

A diferencia de la instrucción de Igualdad esta instrucción se activa siempre y cuando las variables sourceA y sourceB sean completamente distintas. La instrucción NEQ implementada en AuroraLD Studio se muestra en la Figura 115.

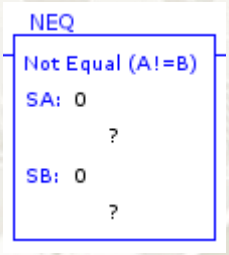


Figura 115: Instrucción NEQ

El funcionamiento básico de esta instrucción se puede representar mediante la siguiente ecuación:

$$SA \neq SB \rightarrow Activates$$

La clase implementada para esta instrucción lleva por nombre NOT_EQUAL. Los atributos correspondientes a esta clase ya fueron declarados y detallados con anterioridad.

A partir de la clase NOT_EQUAL se declaran los objetos denominados NEQ tal como se muestra en la Figura 116.

```
//NOT EQUAL
NOT_EQUAL NEQ[] = new NOT_EQUAL[200];
int neq_counter;
```

Figura 116: Declaración de objetos NEQ

3.4.3.4. Menor que (LES)

Esta instrucción permite comparar dos valores cuyas variables se denominan sourceA y sourceB. Se activa siempre y cuando sourceA tenga un valor inferior a sourceB. La instrucción implementada se muestra en la Figura 117.

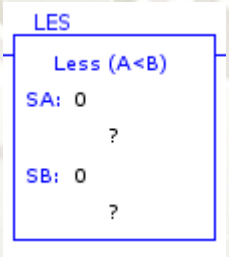


Figura 117: Instrucción LES

La activación de esta instrucción se puede resumir en la siguiente ecuación:

$$SA < SB \rightarrow Activates$$

La clase implementada para esta instrucción lleva por nombre LESS. A partir de la clase LESS se declaran los objetos denominados LES definiendo así un máximo de 200 objetos tal como se muestra en la Figura 118.

```
//LESS
LESS LES[] = new LESS[200];
int les_counter;
```

Figura 118: Declaración de objetos LES

3.4.3.5. Mayor que (GRT)

Esta instrucción es completamente opuesta a la instrucción LES puesto que verifica que la variable sourceB sea menor que sourceA. Por lo tanto, la variable de mayor valor será sourceA en esta instrucción. La instrucción GRT implementada en AuroraLD Studio se muestra en la Figura 119.

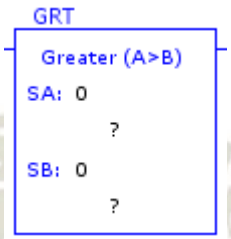


Figura 119: Instrucción GRT

La activación de la instrucción se puede representar mediante la siguiente ecuación:

$$SA > SB \rightarrow Activates$$

La clase implementada para esta instrucción lleva por nombre GREATER. A partir de la clase GREATER se declaran los objetos denominados GRT definiendo un máximo de 200 objetos tal como se muestra en la Figura 120.

```
//GREATER
GREATER GRT[] = new GREATER[200];
int grt_counter;
```

Figura 120: Declaración de objetos GRT

3.4.3.6. Menor o igual que (LEQ)

Esta instrucción comprueba que la variable sourceA sea menor o igual que la variable sourceB para que esta instrucción se pueda activar. La instrucción implementada en AuroraLD Studio se muestra en la Figura 121.

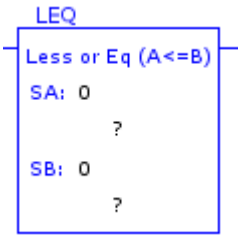


Figura 121: Instrucción LEQ

Básicamente esta instrucción debe satisfacer la siguiente ecuación para poder activarse:

$$SA \leq SB \rightarrow Activates$$

La clase implementada para esta instrucción lleva por nombre LESS_EQ. A partir de la clase LESS_EQ se declaran los objetos denominados LEQ tal como se muestra en la Figura 122. En el código presentado se determina un máximo de 200 objetos para la clase especificada.

```
//LESS EQUAL
LESS_EQ LEQ[] = new LESS_EQ[200];
int leq_counter;
```

Figura 122: Declaración de objetos LEQ

3.4.3.7. Mayor o igual que (GEQ)

La instrucción Mayor o Igual que emplea dos variables las cuales son sourceA y sourceB. Esta instrucción comprueba si el valor de sourceA es mayor o igual que el valor de sourceB. Si la condición especificada se cumple la instrucción se activa. La instrucción implementada en AuroraLD Studio se muestra en la Figura 123.

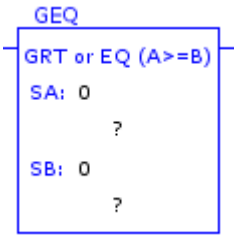


Figura 123: Instrucción GEQ

En otras palabras, se activa la siguiente instrucción si se satisface la siguiente ecuación:

$$SA \geq SB \rightarrow Activates$$

La clase implementada para esta instrucción lleva por nombre GREATER_EQ. A partir de la clase GREATER_EQ se declaran los objetos denominados GEQ tal como se muestra en la Figura 124.

```
//GREATER EQUAL
GREATER_EQ GEQ[] = new GREATER_EQ[200];
int geq_counter;
```

Figura 124: Declaración de objetos GEQ

3.4.4. Instrucciones Matemáticas

Las instrucciones contempladas en el apartado de Funciones Matemáticas son consideradas instrucciones de salida en AuroraLD Studio. Se accede a estas instrucciones al hacer click en la pestaña cuyo nombre es Compute & Math. Dentro de esta categoría se tiene 10 instrucciones diferentes, las instrucciones implementadas se muestran en la Figura 125.

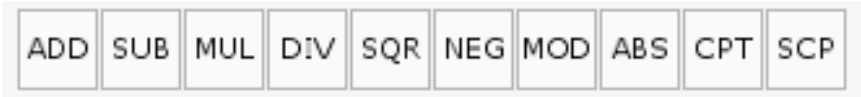


Figura 125: Instrucciones de Funciones Matemáticas

En AuroraLD Studio cada una de las instrucciones matemáticas están representadas por una clase. Las instrucciones ADD, SUB, MUL, DIV y MOD comparten los mismos atributos de clase. Estos atributos se muestran en la Figura 126 así como también el tipo de variable asignado a cada atributo dentro del entorno de processsing.

```
String type;
int num;
String sourceA_s;
String sourceB_s;
String dest_s;
float sourceA;
float sourceB;
float dest;
boolean state;
int belong;
```

Figura 126: Atributos de clase de las instrucciones ADD, SUB, MUL, DIV y MOD

- **type:** Atributo tipo string que hace referencia al tipo de instrucción. En este caso puede ser ADD, SUB, MUL, DIV o MOD dependiendo de la instrucción.
- **num:** Indicador de número de instrucción. Este valor es asignado cada vez que se crea un objeto y sirve para identificar una determinada instrucción.
- **sourceA_s:** Tag asociado a la variable que se encuentra en sourceA (primer valor).
- **sourceB_s:** Tag asociado a la variable que se encuentra en sourceB (segundo valor).
- **dest_s:** Tag asociado a la variable calculada dependiente de sourceA y sourceB de acuerdo al tipo de operación. En otras palabras, hace referencia al resultado.
- **sourceA:** Atributo de tipo float en donde se almacena el primer valor.
- **sourceB:** Atributo de tipo float en donde se almacena el segundo valor.

- **dest:** Atributo de tipo float en donde se almacena el resultado de sourceA y sourceB dependiendo de la operación.
- **state:** Hace referencia al estado de la instrucción ya que podría o no estar activa. Es un atributo de tipo booleano.
- **belong:** Indica la pertenencia de la instrucción a una determinada línea (Rung).

3.4.4.1. Adición (ADD)

La instrucción de adición permite realizar la suma entre dos variables denominadas sourceA y sourceB guardando el resultado en una tercera variable denominada Destination. Esta operación se ejecuta siempre y cuando las condiciones de entrada se cumplan. La instrucción ADD que se ha implementado en AuroraLD Studio se muestra en la Figura 127.

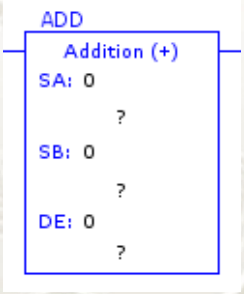


Figura 127: Instrucción ADD

Se puede representar el funcionamiento de esta instrucción mediante la siguiente ecuación:

$$DEST = SRA + SRB$$

La clase creada para esta instrucción se denomina ADDITION y se declaran hasta máximo 200 objetos denominados ADD tal como se muestra en la Figura 128.

```
//ADD
ADDITION add[] = new ADDITION[200];
int add_counter;
```

Figura 128: Declaración de objetos ADD

3.4.4.2. Sustracción (SUB)

La instrucción SUB permite calcular la resta entre dos variables denominadas sourceA y sourceB y guardar este resultado en una variable denominada Destination. La instrucción implementada en AuroraLD Studio se muestra en la Figura 129.

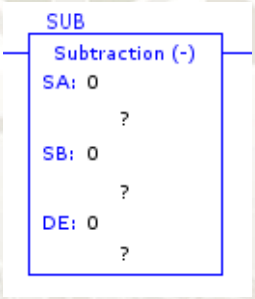


Figura 129: Instrucción SUB

Básicamente funciona en base a la siguiente ecuación:

$$DEST = SRA - SRB$$

La clase creada para esta instrucción se denomina SUBTRACTION y se declaran hasta máximo 200 objetos denominados SUB tal como se muestra en la Figura 130.

```
//SUBTRACTION
SUBTRACTION SUB[] = new SUBTRACTION[200];
int sub_counter;
```

Figura 130: Declaración de objetos SUB

3.4.4.3. Multiplicación (MUL)

Una instrucción MUL permite al usuario realizar una multiplicación entre dos valores cuyas variables en AuroraLD Studio se denominan sourceA y sourceB. El resultado de esta multiplicación se guarda en una variable denominada Destination. La instrucción implementada en AuroraLD Studio se muestra en la Figura 131.

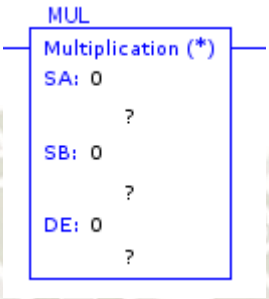


Figura 131: Instrucción MUL

Cuando esta instrucción se activa opera básicamente bajo la siguiente ecuación:

$$DEST = SRA * SRB$$

La clase creada para esta instrucción se denomina MULTIPLICATION y se declaran hasta máximo 200 objetos denominados MUL tal como se muestra en la Figura 132.

```
//MULTIPLICATION
MULTIPLICATION MUL[] = new MULTIPLICATION[200];
int mul_counter;
```

Figura 132: Declaración de objetos MUL

3.4.4.4. División (DIV)

La instrucción de División permite realizar una división entre un valor sourceA y sourceB, donde sourceA es el dividendo y sourceB es el divisor.

El resultado de la división es almacenado y guardado en una variable denominada Destination y es tratada por AuroraLD Studio como una variable global, permitiendo así a otras instrucciones acceder a dicho valor. En la Figura 133 se muestra la instrucción DIV implementada en AuroraLD Studio.

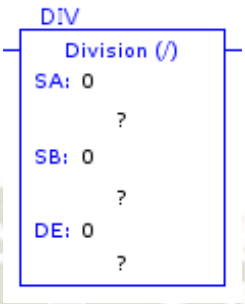


Figura 133: Instrucción DIV

En otras palabras, la instrucción DIV funciona en base a la siguiente ecuación:

$$DEST = \frac{SRA}{SRB}$$

La clase creada para esta instrucción se denomina DIVISION y se declaran hasta máximo 200 objetos denominados DIV tal como se muestra en la Figura 134.

```
//DIVISON
DIVISION DIV[] = new DIVISION[200];
int div_counter;
```

Figura 134: Declaración de objetos DIV

3.4.4.5. Raíz Cuadrada (SQR)

La instrucción SQR permite calcular la raíz cuadra de un valor no negativo. El valor de entrada pasa directamente a la variable sourceA y el resultado es guardado en una variable denominada Destination. En la Figura 135 se muestra la instrucción SQR implementada en AuroraLD Studio.

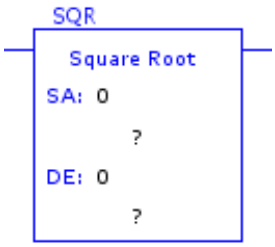


Figura 135: Instrucción SQR

Cada vez que satisfacen las condiciones de entrada esta instrucción se activa y realiza el cálculo de la raíz cuadrada en base a la siguiente ecuación:

$$DEST = \sqrt{SRA}$$

Esta instrucción solo cuenta con un único valor de entrada almacenado en la variable sourceA por lo que los atributos de su clase son similares a los de la instrucción MUL sin considerar aquellos atributos que guardan relación con un segundo valor (source B).

La clase creada para esta instrucción se denomina SQUARE_ROOT y se declaran hasta máximo 200 objetos denominados SQR tal como se muestran en la Figura 136.

```
//SQUARE ROOT
SQUARE_ROOT SQR[] = new SQUARE_ROOT[200];
int sqr_counter;
```

Figura 136: Declaración de objetos SQR

3.4.4.6. Negación (NEG)

Esta instrucción multiplica el valor de entrada por -1 teniendo de esta forma siempre en la salida un valor de entrada negado. Esta instrucción solo contempla dos variables: una de entrada denominada sourceA y una de salida denominada Destination. En la Figura 137 se muestra la instrucción NEG implementada en AuroraLD Studio.

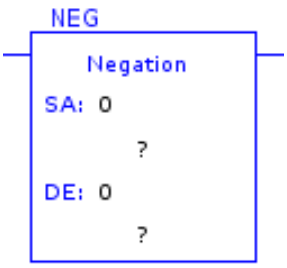


Figura 137: Instrucción NEG

El funcionamiento de esta instrucción se puede representar con la siguiente ecuación:

$$DEST = -SRA$$

Esta instrucción solo cuenta con un único valor de entrada almacenado en la variable sourceA por lo que los atributos de su clase son similares a los de la instrucción MUL sin considerar aquellos atributos que guardan relación con un segundo valor (sourceB).

La clase creada para esta instrucción se denomina NEGATION y se declaran hasta máximo 200 objetos denominados NEG tal como se muestra en la Figura 138.

```
//NEGATION
NEGATION NEG[] = new NEGATION[200];
int neg_counter;
```

Figura 138: Declaración de objetos NEG

3.4.4.7. Módulo o Residuo (MOD)

Cuando se realiza una división entre dos valores algunas veces resulta importante conocer el residuo producto de esa división y para ello se emplea la instrucción MOD. La instrucción MOD que se ha implementado en AuroraLD Studio se muestra en la Figura 139.

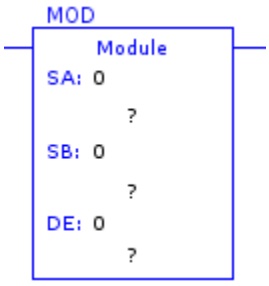


Figura 139: Instrucción MOD

Esta instrucción está basada en una clase denominada MODULE cuyos atributos son idénticos a los encontrados en una instrucción ADD. Basados en la clase MODULE se declaran como máximo 200 objetos denominados MOD tal como se muestran en la Figura 140.

```
//MODULE
MODULE MOD[] = new MODULE[200];
int mod_counter;
```

Figura 140: Declaración de objetos MOD

3.4.4.8. Valor Absoluto (ABS)

La instrucción ABS calcula el valor absoluto de un valor de entrada almacenado en una variable denominada sourceA. El resultado de la operación realizada es guardado en una variable denominada Destination. La instrucción ABS que se ha implementado en AuroraLD Studio se muestra en la Figura 141.

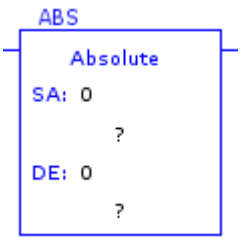


Figura 141: Instrucción ABS

La ecuación que representa esta instrucción es la siguiente:

$$DEST = |SRA|$$

Esta instrucción solo cuenta con un único valor de entrada almacenado en la variable sourceA por lo que los atributos de su clase son similares al de la instrucción MUL sin considerar aquellos atributos que guardan relación con un segundo valor (sourceB).

La clase creada para esta instrucción se denomina ABSOLUTE y se declaran hasta máximo 200 objetos denominados ABS tal cual se muestra en la Figura 142.

```
//ABSOLUTE
ABSOLUTE ABS[] = new ABSOLUTE[200];
int abs_counter;
```

Figura 142: Declaración de objetos ABS

3.4.4.9. Compute (CPT)

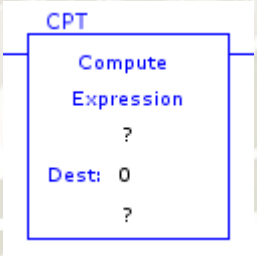


Figura 143: Instrucción CPT

La instrucción CPT mostrada en la Figura 143 es capaz de calcular casi todo tipo de instrucciones matemáticas.

Esta instrucción trabaja en base la librería QScript la cuál es un evaluador de expresiones matemáticas. De todos los operadores y funciones implementadas en la librería QScript, AuroraLD Studio solo soporta los mostrados en la Tabla 8.

Tabla 8: Instrucción CPT Operadores, Funciones y Constantes, Adaptado de la página web QScript

Operadores	
Operador	**Descripción**
+	Adición
-	Sustracción
*	Multiplicación
/	División
^	Potenciación
Funciones	
Función	**Descripción**
abs(a)	Retorna el valor absoluto de a
Int(a)	Retorna el valor entero de un valor a
log(a, b)	Retorna el logaritmo de a en base b
log10(a)	Retorna el logaritmo de a en base 10
logE(a)	Retorna el logaritmo natural de a en base e
rnd(a, b)	Retorna un valor al azar comprendido entre >= a y <b
round(a)	Retorna el entero más cercano de a
signum(a)	Retorna -1 Si a <0, 1 Si a>0 y 0 Si a==0
sqrt(a)	Retorna la raíz cuadra de a. Retorna 0 para valores negativos de a.
acos(a)	Retorna el arcocoseno de a en radianes.
asin(a)	Retorna el arcoseno de a en radianes.
atan(a)	Retorna el arcotangente de a en radianes.
atan2(y,x)	Retorna el ángulo *theta* de la conversión rectangular de las coordenadas (x, y) a coordenadas polares (r, theta) en radianes.
cos(a)	Retorna el coseno de a.
cosh(a)	Retorna el coseno hiperbólico de a.
sin(a)	Retorna el seno de a.
sinh(a)	Retorna el seno hiperbólico de a.
tan(a)	Retorna la tangente de a.
tanh(a)	Retorna la tangente hiperbólica de a.
radians(a)	Convierte un ángulo sexagesimal a radianes.

degrees(a)	Convierte un ángulo en radianes a grados sexagesimales.
Constantes	
Constante	**Descripción**
E	e, La base de logaritmo natural o número de Euler (2.71828...)
PI	Π , Ratio entre la circunferencia de un círculo y su diámetro (3.14…)

La instrucción CPT contempla dos variables importantes para su funcionamiento las cuales son Expression y Destination. En la variable Expression se coloca la expresión a evaluar y su resultado es calculado y almacenado en la variable Destination. Es importante mencionar que las expresiones que se coloquen deben de ser colocadas respetando las mayúsculas y minúsculas puesto que la librería es sensible a esto.

En la Figura 144 se observa la ventana de parámetros implementada en AuroraLD Studio para la instrucción CPT. En esta ventana se puede agregar la expresión matemática a evaluar, así como también el tag donde se guardará la respuesta.

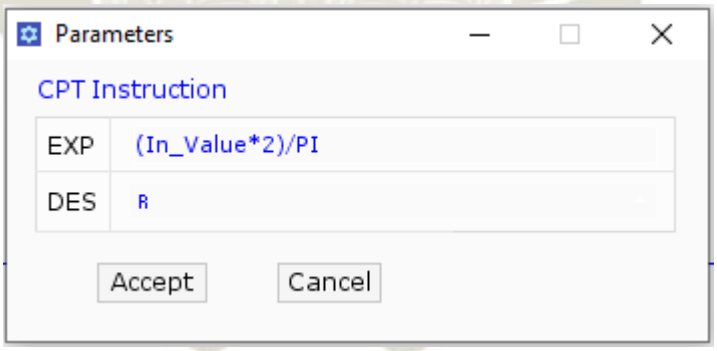


Figura 144: Ejemplo de parámetros Instrucción CPT

La implementación de esta instrucción se sustenta en una clase denominada COMPUTE cuyos atributos se muestran en la Figura 145. Se puede observar en el código también el tipo de variable asignado a cada uno de los atributos. A continuación, se detalla cada uno de estos atributos.

```
class COMPUTE {
  String type;
  int num;
  int x;
  float y;
  String exp_s;
  String dest_s;
  String function;
  float dest;
  boolean state;
  int belong;
```

Figura 145: Atributos de la clase COMPUTE

- **type:** Variable relacionado con el tipo de instrucción. En el caso de la clase COMPUTE el valor de type será CPT.
- **num:** Identificador de la instrucción CPT. A medida que los objetos CPT se crean se les asigna un valor en orden ascendente para posteriormente poder identificarlos y modificar algunos de sus atributos en caso sea necesario.
- **x:** Indica la posición X en pixeles de la instrucción.
- **y:** Indica la posición Y en pixeles de la instrucción.
- **exp_s:** Atributo de tipo String que almacena la expresión matemática a ser evaluada. En algunos casos donde la expresión resulta larga simplemente se tomará los primeros caracteres y se agregarán 3 puntos seguidos.
- **dest_s:** Atributo de tipo string que hace referencia al tag donde se almacenará el valor calculado tras evaluar la expresión matemática.

- **function:** Atributo de tipo string que almacena la expresión matemática a ser evaluada sin importar el tamaño de la expresión a diferencia del atributo exp_s donde se recorta el string.
- **dest:** Atributo de tipo float donde se almacena el valor numérico calculado tras evaluar la expresión.
- **state:** Indica si la instrucción se encuentra activa o no para iniciar la evaluación de la expresión matemática.
- **belong:** Atributo que indica la pertenencia de una instrucción a una determinada línea (Rung).

En base a la clase COMPUTE implementada se declaran como máximo 200 objetos denominados CPT tal cual se muestra en la Figura 146.

```
//COMPUTE
COMPUTE CPT[] = new COMPUTE[200];
int cpt_counter;
```

Figura 146: Declaración de objetos CPT

3.4.4.10. Escalar con parámetros (SCP)

La instrucción SCP permite escalar valores de un determinado rango a otros valores en un rango completamente distinto. Esto es posible realizar al linealizar los valores de entrada conociendo su máximo y mínimo y convirtiéndolos a los valores máximos y mínimos en el rango de salida deseado. En la Figura 147 se observa la instrucción SCP que se ha implementado en AuroraLD Studio.

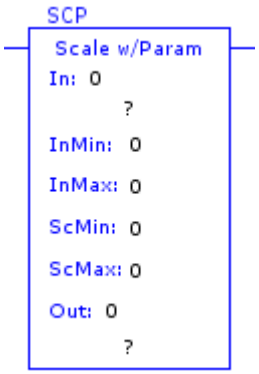


Figura 147: Instrucción SCP

El funcionamiento de esta instrucción se explica con más detalle a través del siguiente ejemplo:

Supongamos que se desea usar en un proceso un sensor de presión cuya salida es analógica y fluctúa entre 0-10v de forma lineal. Además, se conoce que se tiene una tensión de 0v cuando la presión es de aproximadamente 30kPa y una señal de 10v cuando la presión es 150kPa. Al realizar la conexión del sensor e integrarlo en el PLC no se tendrán valores relacionados a la tensión o a la presión, pero si valores enteros producto de la conversión Analógico/Digital. Si la resolución del PLC en el convertidor A/D es de 12 bits entonces se tendrían los siguientes datos:

Tabla 9: Datos del ejemplo SCP

Tensión	Presión	Entrada PLC
0v	30kPa	0
10v	150kPa	4095

Con los datos de la Tabla 9 conocemos que al conectar el sensor al PLC este nos dará valor comprendido entre 0 y 4095 pero esto no lo podemos visualizar como presión o tensión. Para poder hacer esto es necesario linealizar estos parámetros teniendo en el eje X la entrada al PLC y en el eje Y la tensión o presión.

Para dar solución al problema planteado se requiere emplear la ecuación matemática que representa una recta de forma que se pueda encontrar el valor de la presión o temperatura teniendo como parámetro de entrada la conversión analógica/digital.

$$y = mx + b$$

Encontramos la pendiente:

$$m_p = \frac{y_2 - y_1}{x_2 - x_1} \rightarrow m_p = \frac{150 - 30}{4095 - 0} \rightarrow m_p = 0.029304$$

Encontramos el intercepto-y:

$$150 = 0.029304 * 4095 + b_p \rightarrow b_p = 30$$

Finalmente tenemos la ecuación que representa la presión en función de la entrada al PLC:

$$P = 0.029304 * Input + 30$$

Todo el procedimiento descrito es ejecutado por la instrucción SCP de forma automática cada que vez que se cumplen las condiciones de entrada. La ventaja de usar esta instrucción es que solo requiere los parámetros de la tabla para calcular lo demás. Por otro lado, también es posible calcular lo mostrado en el ejemplo empleando la instrucción CPT, pero para poder hacerlo se requiere hacer todo el cálculo de forma manual y colocar la expresión para que pueda ser evaluada.

La implementación de la instrucción SCP se hace en base a la clase SCALE_PARAMETERS cuyos atributos se muestran en la Figura 148. En esta figura se puede observar parte del código necesario para declarar los atributos dentro de la clase.

```
class SCALE_PARAMETERS {
  String type;
  int num;
  int x;
  float y;
  String input_s;
  String output_s;
  float input;
  float output;
  float input_min;
  float input_max;
  float scaled_min;
  float scaled_max;
  boolean state;
  int belong;
```

Figura 148: Atributos de clase SCALE_PARAMETERS

- **type:** Atributo que hace referencia al tipo de instrucción. En este caso su valor sería SCP.
- **num:** Identificador de la instrucción. Este valor va creciendo a medida que se agreguen más instrucciones SCP.
- **x:** Posición X de la instrucción en pixeles.
- **y:** Posición Y de la instrucción en pixeles.
- **input_s:** Tag que hace referencia a la variable de entrada para ser linealizada.
- **output_s:** Tag que hace referencia a la variable de salida para ser linealizada.
- **input:** Variable de tipo float que contiene el valor de la variable de entrada.
- **output:** Atributo que contiene el dato de salida linealizado.

- **input_min:** Valor mínimo de entrada.
- **input_max:** Valor máximo de entrada.
- **scaled_min:** Valor mínimo de salida.
- **scaled_max:** Valor máximo de salida.
- **state:** Atributo que indica el estado de la instrucción (activo o no activo). Se trata de una variable booleana.
- **belong:** Atributo que indica la pertenencia de la instrucción a una determinada línea (Rung).

En base a la clase implementada se declaran los objetos denominados SCP restringiendo como máximo 200 instrucciones de este tipo tal como se muestra en la Figura 149.

```
//SCALE PARAMETERS
SCALE_PARAMETERS SCP[] = new SCALE_PARAMETERS[200];
int scp_counter;
```

Figura 149: Declaración de objetos SCP

3.4.5. Funciones Move/Logical

En AuroraLD Studio se cuenta con una única instrucción en esta categoría la cual es MOV. Se accede a esta instrucción al hacer click en la pestaña Move/Logical. Se considera a la instrucción que se encuentra en esta categoría como una instrucción de salida. En la Figura 150 se muestra la instrucción MOV dentro de la pestaña de instrucciones Move/Logical.

Figura 150: Instrucciones Move/Logical

3.4.5.1. Mover (MOV)

La instrucción MOV es básicamente una instrucción de asignación que permite asignar un valor determinado a una variable escogida por el usuario. Esta instrucción emplea dos variables para su funcionamiento las cuales son sourceA y Destination, siendo sourceA la fuente del valor y Destination la variable escogida por el usuario a la que se le asignará el valor de sourceA. En la Figura 151 se muestra la instrucción MOV que se ha implementado en AuroraLD Studio.

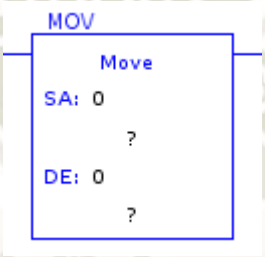


Figura 151: Instrucción MOV

La siguiente ecuación muestra el funcionamiento de la instrucción MOV:

$$DEST = SRA$$

Se implementó la clase denominada MOVE que contiene los atributos mostrados en la Figura 152. En esta figura se muestra parte del código necesario para declarar los atributos dentro de la clase move.

```
class move{
String type;
int num;
int x;
float y;
String sourceA_s;
String dest_s;
float sourceA;
float dest;
boolean state;
int belong;
```

Figura 152: Atributos de clase MOVE

- **type:** Hace referencia al tipo de instrucción. En este caso el valor de la variable type sería MOV.
- **num:** Variable de tipo Int que hace referencia al identificador de cada instrucción MOV. Aumenta el valor conforme se agregan más instrucciones de este tipo.
- **x:** Posición X en pixeles de la instrucción.
- **y:** Posición Y en pixeles de la instrucción.
- **sourceA_s:** Tag que hace referencia al primer valor que será asignado en otra variable distinta.
- **dest_s:** Tag al que le será asignado el valor de la variable representada por el tag sourceA_s.
- **sourceA:** Atributo de tipo float que contiene el valor numérico de entrada.
- **dest:** Atributo de tipo float que guardará el valor asignado.
- **state:** Estado de la instrucción. Al tratarse de una variable booleana puede indicar si se encuentra activa o no la instrucción.

- **belong:** Atributo que hace referencia a la pertenencia de una instrucción a una determinada línea (Rung).

Con la clase MOVE implementada se procede a declarar un máximo de 200 objetos denominados MOV tal como se muestra en la Figura 153.

```
//MOVE
move MOV[] = new move[200];
int mov_counter;
```

Figura 153: Declaración de objetos MOV

3.4.6. Instrucciones Trigonométricas

Las funciones trigonométricas son consideradas en AuroraLD Studio como instrucciones de salida cuyos resultados siempre se asignan y guardan en la variable de Destino (Destination). Estas instrucciones siempre se ejecutan si las condiciones de entrada se cumplen. Se cuenta en total con 6 instrucciones diferentes y se accede a estas instrucciones al hacer click en la pestaña Trigonometric. Las instrucciones implementadas, así como su nombre dentro de AuroraLD Studio se muestran en la Figura 154.

Figura 154: Instrucciones trigonométricas

Todas las instrucciones trigonométricas están basadas en distintas clases, sin embargo, se comparten los mismos atributos que se describen a continuación:

- **type:** Atributo de tipo string que hace referencia al tipo de instrucción. El valor que puede asumir este atributo puede ser cualquiera de las instrucciones trigonométricas: ACS, ASN, ATN, COS, SIN o TAN.

- **num:** Variable de tipo entero que hace referencia al número de instrucción. Mediante esta variable se puede identificar y diferenciar una instrucción de otras de la misma categoría.
- **x:** Posición X en pixeles de la instrucción.
- **y:** Posición Y en pixeles de la instrucción.
- **sourceA_s:** Tag que hace referencia al valor de entrada respecto al cual se calculará el coseno, seno, tangente, etc.
- **dest_s:** Tag que hace referencia al valor de salida. Es en este tag donde guardará el valor calculado.
- **sourceA:** Variable de tipo float que almacena el valor de entrada.
- **dest:** Variable de tipo float que almacena el valor calculado.
- **state:** El atributo estado hace referencia a si se encuentra o no activa la instrucción tras cumplirse o no las condiciones de entrada.
- **belong:** Valor que indica pertenencia de una instrucción a una determinada línea (Rung).

Los atributos previamente descritos se muestran en el código definiendo así los atributos principales de la clase con el tipo correspondiente de variable tal como se muestra en la Figura 155.

```
class ACOS {
  String type;
  int num;
  int x;
  float y;
  String sourceA_s;
  String dest_s;
  float sourceA;
  float dest;
  boolean state;
  int belong;
```

Figura 155: Atributos de clase ACOS

3.4.6.1. Arcocoseno (ACS)

La instrucción ACS permite calcular el arcocoseno de un valor definido en la variable sourceA y cuyo resultado se almacena en la variable Destination. Se muestra la instrucción ACS implementada en AuroraLD Studio en la Figura 156.

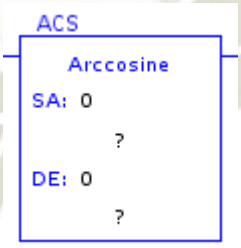


Figura 156: Instrucción ACS

Es posible representar el funcionamiento de la instrucción en base a la siguiente ecuación:

$$DEST = \cos^{-1}(SRA)$$

La clase creada para esta instrucción se denomina ACOS y se declaran hasta máximo 200 objetos denominados ACS tal como se muestra en la Figura 157.

```
//ACOS
ACOS ACS[] = new ACOS[200];
int acs_counter;
```

Figura 157: Declaración de objetos ACS

3.4.6.2. Arcoseno (ASN)

Esta instrucción permite calcular el arcoseno de un valor definido en sourceA. Esta instrucción se ejecuta siempre y cuando se cumplan las condiciones de entrada. La instrucción se ha implementado en AuroraLD studio tal como se muestra en la Figura 158.

Figura 158: Instrucción ASN

Se puede representar el funcionamiento de esta instrucción en base a la siguiente ecuación:

$$DEST = \sin^{-1}(SRA)$$

La clase creada para esta instrucción se denomina ASIN y se declaran hasta máximo 200 objetos denominados ASN tal como se muestra en la Figura 159.

```
//ASINE
ASIN ASN[] = new ASIN[200];
int asn_counter;
```

Figura 159: Declaración de objetos ASN

3.4.6.3. Arcotangente (ATN)

La instrucción ATN calcula el arcotangente de un valor definido en sourceA. Se muestra en la Figura 160 la instrucción ATN implementada en AuroraLD Studio.

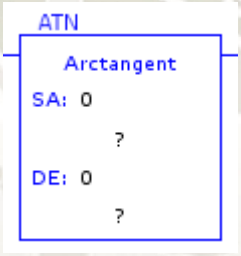


Figura 160: Instrucción ATN

Se representa el funcionamiento de la instrucción en base a la siguiente ecuación:

$$DEST = \tan^{-1}(SRA)$$

La clase creada para esta instrucción se denomina ATAN y se declaran hasta máximo 200 objetos denominados ATN tal cual se observa en el código presentado en la Figura 161.

```
//ATAN
ATAN ATN[] = new ATAN[200];
int atn_counter;
```

Figura 161: Declaración de objetos ATN

3.4.6.4. Coseno (COS)

La instrucción COS calcula el coseno del valor asignado a sourceA. El valor calculado es guardado en la variable Destination la cual es una variable global que permite el acceso a otras instrucciones que deseen utilizar este valor. La instrucción COS que se ha implementado en AuroraLD Studio se muestra en la Figura 162.

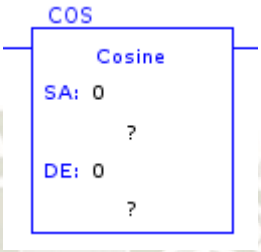


Figura 162: Instrucción COS

El funcionamiento de esta instrucción está representado por la siguiente ecuación:

$$DEST = \cos(SRA)$$

La clase creada para esta instrucción se denomina COSINE y se declaran hasta máximo 200 objetos denominados COS tal como se observa en el código presentado en la Figura 163.

```
//COSINE
COSINE COS[] = new COSINE[200];
int cos_counter;
```

Figura 163: Declaración de objetos COS

3.4.6.5. Seno (SIN)

La instrucción SIN permite al usuario calcular el Seno de un valor definido en sourceA y cuyo resultado se almacena en Destination. La instrucción implementada en AuroraLD Studio se muestra en la Figura 164.

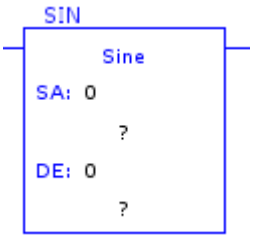


Figura 164: Instrucción SIN

La ecuación que define el funcionamiento de esta instrucción se muestra a continuación:

$$DEST = \sin(SRA)$$

La clase creada para esta instrucción se denomina SINE y se declaran hasta máximo 200 objetos denominados SIN tal como se muestra en la Figura 165.

```
//SINE
SINE SIN[] = new SINE[200];
int sin_counter;
```

Figura 165: Declaración de objetos SIN

3.4.6.6. Tangente (TAN)

La instrucción TAN permite al usuario calcular la Tangente de un valor definido en sourceA y cuyo resultado es guardado en Destination. La instrucción implementada en AuroraLD Studio se muestra en la Figura 166.

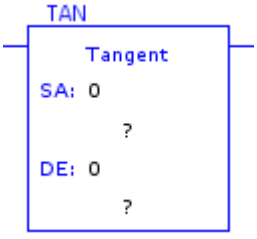


Figura 166: Instrucción TAN

La ecuación que define el funcionamiento de la instrucción TAN se muestra a continuación:

$$DEST = \tan(SRA)$$

La clase creada para esta instrucción se denomina TANE y se declaran hasta máximo 200 objetos denominados TAN tal cual se ve en la Figura 167.

```
//TANE
TANE TAN[] = new TANE[200];
int tan_counter;
```

Figura 167: Declaración de objetos TAN

3.4.7. Instrucciones de Conversión

AuroraLD Studio contempla dos instrucciones de conversión en esta categoría y se accede a estras instrucciones al hacer click en la pestaña Math Conversion. Estas instrucciones están categorizadas como instrucciones de salida por lo que deben cumplirse las condiciones de entrada para que puedan activarse. En la Figura 168 se muestran las instrucciones que se han implementado en AuroraLD Studio referentes a las instrucciones de conversión.

Figura 168: Instrucciones de conversión

Las dos instrucciones de conversión con las que cuenta AuroraLD Studio emplean distintas clases, pero comparten los atributos que se muestran en la Figura 169.

```
class DEGR {
  String type;
  int num;
  int x;
  float y;
  String sourceA_s;
  String dest_s;
  float sourceA;
  float dest;
  boolean state;
  int belong;
```

Figura 169: Atributos de la clase DEGR

- **type:** Atributo que hace referencia al tipo de instrucción. En este caso esta variable puede asumir valores como DEG o RAD.
- **num:** Identificador de la instrucción. Conforme se crean instrucciones aumenta el valor y se le asigna un número único a cada instrucción creada.
- **x:** Posición X en pixeles de la instrucción.
- **y:** Posición Y en pixeles de la instrucción.
- **sourceA_s:** Tag asociado a la variable de entrada que será convertida en ángulos sexagesimales o radianes.
- **dest_s:** Tag asociado a la variable donde se colocará la respuesta tras la conversión.
- **sourceA:** Valor numérico de entrada a ser convertido.
- **dest:** Valor numérico de salida tras la conversión.
- **state:** Indica el estado de la instrucción (activo o no activo).
- **belong:** Atributo que indica pertenencia de una instrucción a una determinada línea (Rung).

3.4.7.1. Radianes a Sexagesimales (DEG)

La instrucción DEG permite la conversión de radianes a grados sexagesimales. El valor de entrada hace referencia a la variable sourceA y la salida (valor convertido) hace referencia a la variable Destination. La instrucción DEG que se ha implementado en AuroraLD Studio se muestra en la Figura 170.

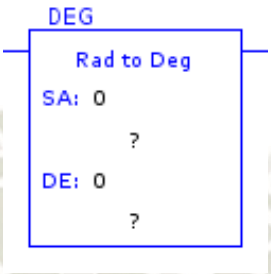


Figura 170: Instrucción DEG

En base a la clase DEGR implementada se declaran como máximo 200 objetos denominados DEG tal como se muestra en la Figura 171.

```
//DEGREES
DEGR DEG[] = new DEGR[200];
int deg_counter;
```

Figura 171:Declaración de objetos DEG

3.4.7.2. Sexagesimales a Radianes (RAD)

La instrucción RAD permite al usuario convertir grados sexagesimales a grados en radianes. El valor de entrada es asignado a sourceA y la salda (conversión) es asignado a la variable Destination. La instrucción implementada en AuroraLD Studio se muestra en la Figura 172.

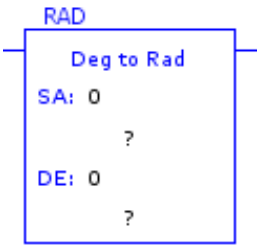


Figura 172: Instrucción RAD

En base a la clase RADIAN implementada se declaran como máximo 200 objetos denominados RAD tal como se muestra en la Figura 173.

```
//RADIANS
RADIAN RAD[] = new RADIAN[200];
int rad_counter;
```

Figura 173: Declaración de objetos RAD

3.4.8. Instrucciones Especiales

Dentro de las instrucciones especiales se cuenta con una única instrucción denominada PID. Se accede a esta categoría al hacer click en la pestaña Special instructions. En la Figura 174 se observa la instrucción contenida en las instrucciones especiales.

Figura 174: Instrucciones especiales

3.4.8.1. Control PID (PID)

La instrucción PID también conocida como control proporcional-integral y derivativo es un control de lazo cerrado en la que una variable de salida es controlada de forma que se aproxime y se mantenga en un valor de entrada denominado Set Point (SP). Esta instrucción es considerada en AuroraLD Studio como una instrucción de salida.

Este tipo de control trabaja con una ganancia proporcional, una ganancia derivativa y una ganancia integral y está representado por la siguiente ecuación:

$$CV(t) = kp * e(t) + ki * \int e(t)\, dt + kd * \frac{de(t)}{dt}$$

Suponiendo que se quiere controlar la temperatura en un determinado ambiente. La entrada se recolectaría en base a los datos del sensor de temperatura y debe existir algún tipo de actuador que caliente o enfríe el ambiente de manera que se tenga una temperatura deseada. Estos actuadores trabajarán cuando sea necesario y a intensidades distintas de acuerdo a la variable de control (CV).

La forma en cómo operan estos actuadores dependerá mucho de los valores de las ganancias proporcional, integral y derivativa. La instrucción PID implementada en AuroraLD Studio se muestra en la Figura 175.

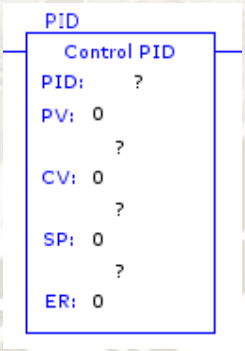


Figura 175: Instrucción PID

Esta instrucción maneja distintos parámetros referentes a:

- **Set point (SP):** El valor de Set Point hace referencia al valor que se desea alcanzar en la variable de proceso y por lo general es determinada por el usuario.
- **Variable de control (CV):** La variable de control hace referencia a la salida del control PID cuya señal se inyecta a los actuadores de manera que ayuden a alcanzar el valor SP.

Los parámetros de esta instrucción se agregan empleando la ventana de parámetros de la instrucción PID. En esta ventana se puede asignar distintos tags a las variables de control correspondientes tal como se muestra en la Figura 176.

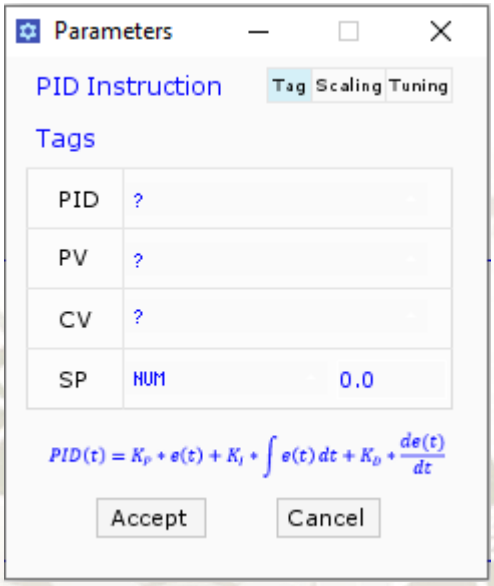


Figura 176: Variables de Instrucción PID

- **Valores no escalados de entrada:** Los valores no escalados de entrada hacen referencia a la entrada de los sensores. Valores que por lo general están expresados en función de la resolución del conversor analógico/digital.
- **Valores escalados de entrada:** Los valores escalados de entrada funcionan como una instrucción SCP ya que permite escalar los valores de entrada en un rango de valores que se requieren para el proceso.
- **Valores escalados de salida:** El valor escalado de salida estará en función de la resolución del conversor digital/analógico.

Las variables previamente detalladas en función del escalamiento se agregan y configuran en la ventana de parámetros de la instrucción PID en la que se asignan tanto los valores no escalados como los valores escalos de entrada y salida. Estas variables se muestran en la Figura 177.

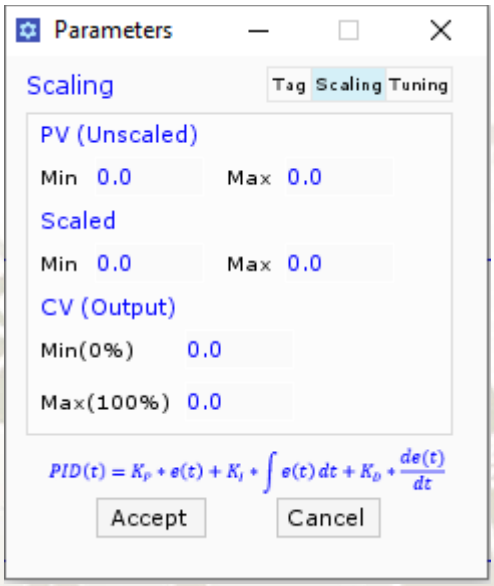


Figura 177: Variables de escalamiento Instrucción PID

- **Ganancias:** Las ganancias hacen referencia a los valores que asumen las constantes Kp, Kd y Ki. De acuerdo a los valores de las ganancias se tendrá un comportamiento distinto en la variable de control haciendo al sistema estable o inestable.
- **Tiempo de muestreo:** El tiempo de muestreo debe escogerse de forma que se tengan suficientes valores para representar la señal que se desea analizar de lo contrario no se tendrá un buen control. El objetivo es evitar el aliasing.
- **Acción de control:** La acción de control puede ser directa o inversa. Cuando incrementa la señal del proceso (PV) por encima del Set Point (SP) y la señal de la variable de control cae hace referencia a un tipo de control inverso. Por otro lado, si la señal PV incrementa por encima de SP y la señal CV también incrementa hace referencia a un control directo.

Para asegurar un funcionamiento adecuado del control PID es necesario sintonizar los parámetros del controlador, para ellos se agregan los valores de las constantes, el tiempo de muestreo y la dirección de control en la ventana de parámetros del control PID tal como se muestra en la Figura 178.

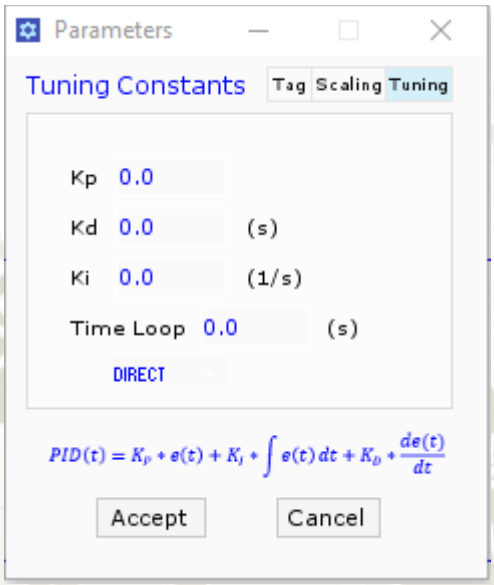


Figura 178: Constantes de sintonización

El funcionamiento de esta instrucción se basa en el entendimiento básico del control PID:

- Se realiza un muestreo cada cierto tiempo por lo que el tiempo de muestreo es un parámetro conocido.
- Se calcula el error al realizar la resta entre el Set Point y la variable de proceso logrando de esta forma determinar el valor del error actual con su respectivo tiempo.
- Se multiplica el valor del error por la ganancia proporcional obteniendo así lo que se conoce como **Control Proporcional.**

$$CV(t) = kp * e(t)$$

- La derivada del error se puede obtener a través de una diferencia y un cociente ya que por definición la derivada se puede aproximar a:

$$\frac{de(t)}{dt} \approx \frac{\Delta e(t)}{\Delta t}$$

- De la ecuación anterior la diferencia de tiempo es un valor conocido ya que es el tiempo de muestreo.
- La diferencia del error se puede obtener haciendo la resta del error actual con el error previo para ello debe de manejarse dos variables: una que almacene el error actual y otra que almacene el error anterior.
- El cociente calculado se multiplica por la ganancia derivativa obtenido así lo que se conoce como **Control Derivativo.**

$$CV(t) = kd * \frac{de(t)}{dt}$$

- La integral del error se calcula encontrando el área bajo la curva ya que por definición el área bajo la curva es la integral.
- Esta área se calcula por aproximación empleando la integración por rectángulos en donde el ancho del rectángulo es el tiempo de muestreo y el alto del rectángulo el valor actual del error.
- Ya que el error cambia en el tiempo se tendrán distintas áreas que deben irse sumando para así al final tener la integral del error.

$$\int e(t)\, dt \approx \sum A$$

- Finalmente se multiplica la integral del error por la ganancia integral obteniendo así lo que se conoce como **Control Integral.** La acción integral no debe actuar en un primer momento y solo entrar en funcionamiento cuando el error se encuentra cerca

de 0, para ello se incrementa de a pocos la variable de control permitiendo así alcanzar el Set Point.

$$CV(t) = ki * \int e(t)\, dt$$

- De la forma planteada y sumando cada uno de los controles, se obtiene lo que se denomina **Control PID Discreto** al estar basado en un tiempo de muestreo determinado.

$$CV(k) = kp * e(k) + ki * \sum A + kd * \frac{\Delta e(k)}{\Delta t}$$

La instrucción PID se implementa en base a una clase denominada CONTROL_PID cuyos atributos se muestran a continuación:

- **name:** Atributo que hace referencia al tag de tipo PID asociado a una determinada instrucción PID.
- **type:** Hace referencia al tipo de instrucción. En este caso esta variable toma el valor de PID.
- **num:** Hace referencia al identificador de la instrucción. Este dato es único e irrepetible e incrementa conforme se agregan más instrucciones PID en el diagrama Ladder.
- **x:** Coordenada X en pixeles de la instrucción PID.
- **y:** Coordenada Y en pixeles de la instrucción PID.
- **state:** Indica el estado de la instrucción, la cual puede ser true o false al tratarse de una variable booleana. La instrucción se activa si las condiciones de entrada se cumplen.
- **belong:** Indica la pertenencia de una instrucción a una determinada línea (Rung).
- **PV_s:** Atributo de tipo String que hace referencia al tag de la variable de proceso.

- **CV_s:** Atributo de tipo String que hace referencia al tag de la variable de control
- **SP_s:** Atributo de tipo String que hace referencia al tag del set point.
- **PV:** Valor numérico de la variable de proceso.
- **CV:** Valor numérico de la variable de control.
- **SP:** Valor numérico de la variable de set point.
- **error:** Valor numérico del error al realizar la diferencia del set point y la variable de proceso.
- **kp:** Variable de tipo float que hace referencia a la ganancia proporcional.
- **kd:** Variable de tipo float que hace referencia a la ganancia derivativa.
- **ki:** Variable de tipo float que hace referencia a la ganancia integral.

```
class CONTROL_PID {
  String name;
  String type;
  int num;
  int x;
  float y;
  boolean state;
  int belong;
  String PV_s;
  String CV_s;
  String SP_s;
  float PV;
  float CV;
  float SP;
  float error;
  float kp;
  float kd;
  float ki;
```

Figura 179: Atributos de la clase CONTROL_PID

En base a la clase implementada se declaran como máximo 200 objetos denominados PID tal como se muestra en la Figura 180.

```
//PID
CONTROL_PID PID[] = new CONTROL_PID[200];
int pid_counter;
```

Figura 180: Declaración de objetos PID

3.5. Espacio que ocupan las instrucciones en el entorno Ladder

Cada una de las instrucciones presentes en AuroraLD Studio tiene un tamaño definido el cual se puede expresar en función de cuantas líneas ocupa. A continuación, se define esta medida en función de la categoría a la que pertenece una instrucción:

a) Comunes

Todas las instrucciones de tipo común ocupan una sola línea por lo que cada vez que son agregadas al espacio de trabajo no generan un desplazamiento del resto de líneas que se encuentren debajo.

b) Temporizadores/Contadores

Las instrucciones de temporizadores y contadores ocupan dos líneas por lo que cada vez que son agregadas generan un desplazamiento del resto de líneas que encuentran debajo. Dentro de esta categoría la instrucción de RESET es una bobina y solo ocupa una sola línea.

c) Comparadores

Todas las instrucciones de comparación ocupan tres líneas del espacio de trabajo Ladder.

d) Funciones Matemáticas

De todas las instrucciones que podemos encontrar en esta categoría la instrucción SCP ocupa 4 líneas y las demás solamente ocupan 3 líneas.

e) Funciones de Asignación

La instrucción MOV ocupa 3 líneas.

f) Funciones Trigonométricas

Todas las instrucciones que se encuentran en esta categoría ocupan 3 líneas del espacio de trabajo Ladder.

g) Funciones de Conversión

Las instrucciones de conversión que se encuentran en esta categoría ocupan 3 líneas.

h) Instrucciones Especiales

La instrucción PID que se encuentra en esta categoría ocupa 4 líneas del espacio de trabajo Ladder.

3.6. Implementación de la tabla de datos del programa

En algunos softwares de programación de PLCs se suele emplear las direcciones de las salidas y/o entradas en las instrucciones Ladder como variables que son asociadas a una lógica determinada cuando el programa se ejecuta. Sin embargo, de acuerdo al estándar IEC 61131-3 las direcciones de un entorno de programación se reemplazan por variables con un nombre y tipo de dato específico. Esto facilita el entendimiento del programa y evita errores al momento de la ejecución del mismo.

En AuroraLD Studio se cuenta con una ventana que permite la creación de Tags para ser usados en el programa. Estos tags se configuran asignando un nombre, tipo de variable y la dirección respectiva. La ventana mostrada en la Figura 181 permite la configuración de los Tags y se denomina Tabla de Datos del Programa también denominado Program Tags.

Program Tags

N	TAG	VARIABLE	ADDRESS	VALUE	DESCRIPTION
0		VAR	PLC	0	
1		VAR	PLC	0	
2		VAR	PLC	0	
3		VAR	PLC	0	
4		VAR	PLC	0	
5		VAR	PLC	0	
6		VAR	PLC	0	

Figura 181: Tabla de datos del programa

El algoritmo implementado permite generar la tabla que se muestra en la Figura 181. Además, la librería control P5 permite crear cuadros de texto y listas desplegables que se emplean en determinadas columnas de la tabla de datos del programa. La máxima cantidad de Tags que se pueden crear en AuroraLD Studio es 100.

Las variables empleadas para crear la tabla de datos de programa se muestran en la Figura 182. En la figura se tienen 5 arrays distintos referentes a cada una de las columnas que se observan en la tabla de datos del programa (Figura 182).

```
//PROGRAM TAGS
boolean PRG_TAGS = false;
int num_tags = 100;
int ptag_counter = -1;
int tag_pressed=-1;
int blank_tag=0;
int future_data=-1;
String tag_name[];
String var_type[];
String plc_dir[];
String value_num[];
String des_name[];
int temporal_var = 0;
```

Figura 182: Variables de la tabla de datos del programa

En el caso de las placas de desarrollo que están soportadas por defecto como en el caso de Arduino Uno y Arduino Nano se tienen los parámetros preconfigurados mostrados en la Figura 183. En el código se observa las direcciones de entrada o salida que tienen asignados las placas de desarrollo.

```
List prg_var = Arrays.asList("Int", "Real", "Bool", "Timer", "Counter", "PID");
List prg_plc_var = Arrays.asList("");
List prg_plc_inre = Arrays.asList("Internal", "AI:3/0", "AI:3/1",
  "AI:3/2", "AI:3/3", "AO:4/0", "AO:4/1", "AO:4/2", "AO:4/3"); //NANO / UNO
List prg_plc_bool = Arrays.asList("Internal", "DI:1/0", "DI:1/1",
  "DI:1/2", "DI:1/3", "DI:1/4", "DI:1/5", "DO:2/0", "DO:2/1", "DO:2/2",
  "DO:2/3", "DO:2/4", "DO:2/5"); // NANO / UNO
List prg_plc_tcp = Arrays.asList("Internal");
```

Figura 183: Tags preconfigurados para Arduino Nano y Arduino Uno

Los tags que se pueden crear en el software están comprendidos por 5 parámetros bien definidos los cuales se pueden describir de la siguiente forma:

Nombre (Tag): Hace referencia al nombre de la etiqueta. Esta etiqueta representará a una dirección determinada. En la Figura 184 se observa un ejemplo de un tag llamado BOMBA_1.

Figura 184: Ejemplo de Tag

Variable (Variable): Existen 6 tipos de variables distintas que pueden ser asignadas a cada uno de los Tags que se crean en el software. El tipo de variable depende mucho de la aplicación de dicho Tag. Se muestran en la Figura 185 los tipos de variables que se pueden emplear en AuroraLD Studio.

Figura 185: Tipos de variables

Los tipos de variables implementados en AuroraLD Studio son:

- **INT:** Esta clase de variable hace referencia a valores enteros.
- **REAL:** Esta clase de variables hace referencia a valores flotantes (decimales).
- **BOOL:** Variables booleanas que se pueden interpretar como Verdadero o Falso, Alto o Bajo, etc.
- **TIMER:** Variable empleada por todos los tipos de temporizadores. Define a su vez la etiqueta correspondiente de cada temporizador.

- **COUNTER:** Variable empleada por todos los tipos de contadores. Define a su vez la etiqueta correspondiente de cada contador.
- **PID:** Esta variable le brinda una etiqueta al bloque PID así todas las variables relacionadas con esta instrucción se pueden diferenciar de otros bloques PID.

Dirección (Address): La dirección de un Tag resulta importante ya que le indica al PLC a que slot y bit se le asignará un determinado Tag. Conociendo los Slots de los PLCs podemos también verificar que tarjetas que tienen instalados y así manejar entradas y salidas digitales o analógicas de forma correcta. En AuroraLD Studio se opta por emplear las direcciones en función del tipo de variable tal como se muestra en la Tabla 10.

Tabla 10: Tipos de Direcciones

Variable	Dirección	Ejemplo
INT	AI, AO, Internal	AI:3/0, AI:3/5, AO:4/2, Internal
REAL	AI, AO, Internal	AI:3/0, AI:3/5, AO:4/2, Internal
BOOL	DI, DO, Internal	DI:1/0, DO:2/3, Internal
TIMER	Internal	Internal
COUNTER	Internal	Internal
PID	Internal	Internal

- Siempre que se trate de una entrada digital se empleará las letras DI haciendo referencia a Digital Input y siempre estará ubicado en el Slot 1.
- Siempre que se trate de una salida digital se empleará las letras DO haciendo referencia a Digital Output y siempre estará ubicado en el Slot 2.
- Siempre que se trate de una entrada analógica se empleará las letras AI haciendo referencia a Analog Input y siempre estará ubicado en el Slot 3.
- Siempre que se trate de una salida analógica se empleará las letras AO haciendo referencia a Analog Output y siempre estará ubicado en el Slot 4.

- Finalmente, todas aquellas variables cuyas direcciones sean Internas son variables manejadas internamente por cada una de las instrucciones en donde sean asignadas. Este tipo de dirección funciona para todos los tipos de variables.

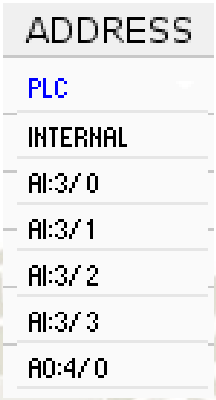


Figura 186: Direcciones de Tags

Valor (Value): El valor de un Tag se define de dos formas la primera forma es automática y los valores se calculan en función de la lógica Ladder implementada para todos aquellos Tags que no tengan una dirección del tipo Internal. Por otro lado, aquellos Tags que han sido declarados con una dirección del tipo Internal sus valores no se generan de forma automática ya que permiten ingreso de datos manuales.

En conclusión, si se desea manipular un dato mientras el programa se encuentra en ejecución es mejor emplear un tipo de dirección Internal, pero si se desea visualizar datos provenientes del PLC o datos calculados en base a la lógica implementada se debe emplear otro tipo de dirección que no sea Internal. Por defecto los tags tendrán un valor de 0 como se muestra en la Figura 187.

Figura 187: Ejemplo de valor (Tag)

Descripción (Description): La descripción de un Tag se considera opcional puesto que no es necesario para el funcionamiento, sin embargo, ayuda al entendimiento del programa ya que es más fácil identificar un Tag y su función en la lógica Ladder. Un ejemplo de descripción de un tag se muestra en la Figura 188.

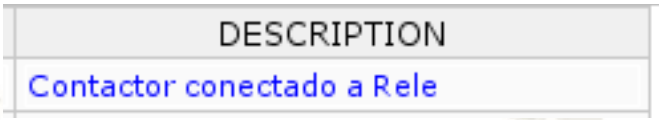


Figura 188: Descripción de un tag

3.7. Leer y escribir datos en cada una de las instrucciones

La lectura siempre se realiza obteniendo los datos de la Tabla de Datos del Programa y la escritura de los datos siempre se realiza pasando los datos a la Tabla de Datos del Programa tal y como se detalló en la arquitectura del software. De esta forma se asegura que los datos que se puedan calcular en cada una de las instrucciones sean globales y se puedan compartir entre instrucciones.

Todas las instrucciones Ladder cuentan con un método que permite la lectura y escritura de datos. El método empleado lleva por nombre data() y el algoritmo implementado para realizar la lectura y escritura es similar en todas las instrucciones y se muestra en la Figura 189.

```
void data() {

  if (row_tag!=-1) {
    if (var_type[row_tag].equals("Bool")) {
      if (int(value_num[row_tag])==0) {
        state=false;
        value=0;
      } else if (int(value_num[row_tag])==1) {
        state=true;
        value=1;
      }
    } else if ((name.substring(name.length()-2, name.length())).equals("DN")) {
      if (val_DN[row_tag]==0) {
        state=false;
        value=0;
      } else if (val_DN[row_tag]==1) {
        state=true;
        value=1;
      }
```

```
      } else if ((name.substring(name.length()-2, name.length())).equals("EN")) {
        if (val_EN[row_tag]==0) {
          state=false;
          value=0;
        } else if (val_EN[row_tag]==1) {
          state=true;
          value=1;
        }
      }
    }
  }
}
```

Figura 189: Algoritmo data()

En el algoritmo de la Figura 189 se inicia evaluando el atributo row_tag el cual hace referencia a la fila en la que se encuentra el tag asociado a una determinada instrucción. El valor de row_tag es diferente de -1 si tiene un tag asociado. Luego se comprueba el tipo de variable asociado a ese Tag y en función de eso se escriben los datos en la Tabla de Datos del Programa (Program Tags).

3.8. Métodos de cada instrucción

Los métodos de las instrucciones hacen referencia a las funciones implementadas en cada una de las clases, pero en este apartado se dará más importancia a aquella función que permite el cálculo (funcionamiento) de las instrucciones las cuales al ser llamadas ejecutan y realizan la función que le corresponde a cada instrucción.

La mayoría de instrucciones Ladder cuentan con una función que realiza algún cálculo. Esta función solo se activa y puede ser llamada cuando el programa (en su conjunto) entra en modo de ejecución. Este método dentro de cada una de las instrucciones se denomina calc().

El método calc() toma los atributos de la clase para realizar los cálculos respectivos. En la Figura 190 se muestra un ejemplo de código que permite realizar la cuenta dentro de la instrucción CTU. En el código observamos que aparte de la cuenta también se tiene la lógica necesaria para el resto de variables asociadas al contador CTU tales como: DN y EN.

```
void calc() {

  if (state==true) {
    en=true;
    if (cont==0) {
      acc++;
      cont=1;
    }
  } else if (state==false) {
    en=false;
    cont=0;
  }

  if (acc>=preset) {
    dn=true;
  } else {
    dn=false;
  }
}
```

Figura 190: Método calc() de una instrucción CTU

En el algoritmo podemos ver cómo se incrementa el acumulador cada vez que cambia el estado de la instrucción realizando así la cuenta. En el momento en el que el Acumulador alcanza el valor de Preset se activa la variable interna del contador DN (Done).

3.9. Salidas del diagrama Ladder

A medida que se han ido creando las instrucciones para el software AuroraLD Studio se ha ido definiendo las instrucciones que se consideran como Salidas las cuales son denominadas Saving según el estándar IEC 61131-3.

Las instrucciones de salida solo son 30 de las 40 instrucciones con las que cuenta el software y se listan a continuación:

- BST, OTE, OTL, OTU, TON, TOF, RTO, CTU, CTD, RES, ADD, SUB, MUL, DIV, SQR, NEG, MOD, ABS, CPT, SCP, MOV, ACS, ASN, ATN, COS, SIN, TAN, DEG, RAG, PID

En AuroraLD Studio las instrucciones de salida siempre se colocan en el lado derecho del diagrama Ladder y siempre deben de cumplirse las condiciones de entrada para que estas puedan ser activadas. Basado en lo mencionado anteriormente, notamos que el algoritmo a implementar es bastante directo ya que si se cumplen las condiciones de entrada se debe llamar al método de cálculo correspondiente a la instrucción de salida que se está evaluando.

El algoritmo no solo debe de ejecutarse para una línea, sino que debe hacerse de forma cíclica y siempre de arriba hacia abajo línea a línea. Además, para asegurar de que una instrucción de salida que no corresponde a una línea se active se hace uso del atributo que indica pertenencia. Dicho atributo de pertenencia se encuentra en todas las clases con el nombre Belong, siempre es un valor entero e indica a que línea pertenece una determinada instrucción.

```
//OUTPUTS
////////////////////
//OTE
for (int i=0; i<ote_counter; i++) {
  if (OTE[i].belong==u) {
    if (rung_state[u]==true) {
      OTE[i].state=true;
    } else {
      OTE[i].state=false;
    }
  }
  OTE[i].data();
}
```

Figura 191: Algoritmo de ejecución de salida OTE

En el caso de la instrucción OTE podemos observar en la Figura 191 que se comprueba la pertenencia de la instrucción con respecto a línea que se está evaluando. Además, se verifica el cumplimiento de las instrucciones de entrada empleando la variable rung_state. Si la variable rung_state es Verdadera se procede con la ejecución de las instrucciones de salida.

En la Figura 191 se ha presentado un ejemplo para la instrucción OTE, pero en el resto de instrucciones el algoritmo opera de forma similar.

Para ayudar al entendimiento del funcionamiento del algoritmo de ejecución de las instrucciones de salida se muestra en la Figura 192 un diagrama de flujo en el que se describe de forma general el proceso de ejecución de las salidas.

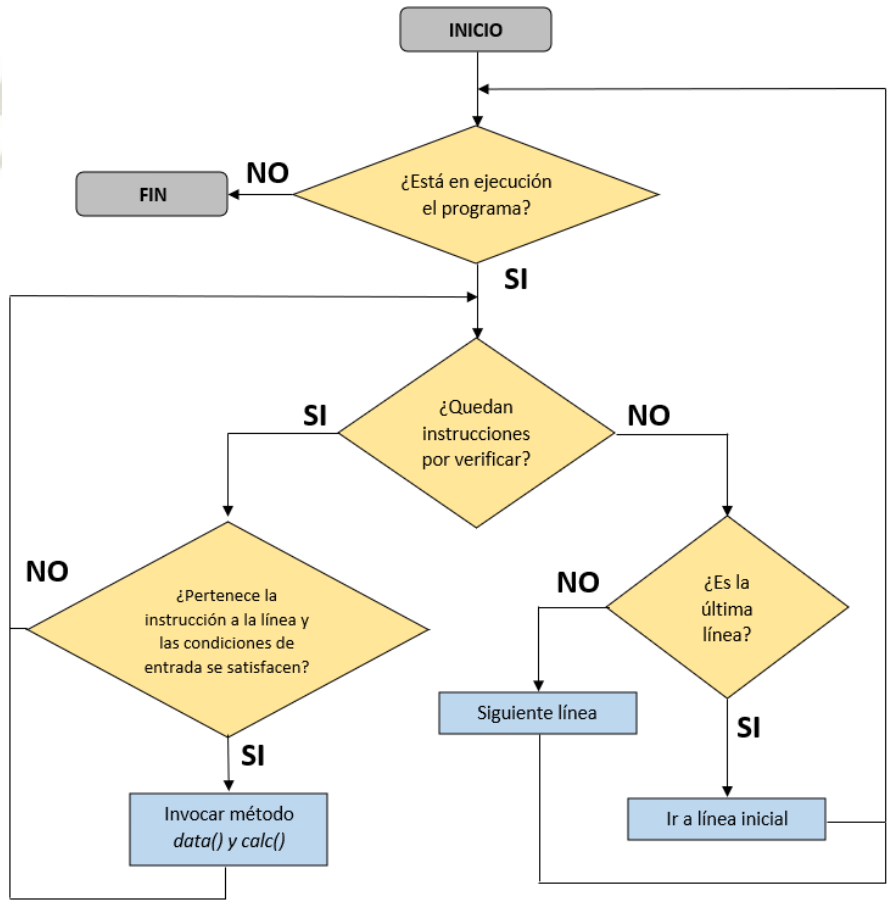


Figura 192: Diagrama de flujo de la ejecución de las instrucciones de salida

3.10. Entradas del diagrama Ladder

Las instrucciones de entrada también son denominadas según el estándar IEC 61131-3 como Computing. En AuroraLD Studio las instrucciones de entrada se listan a continuación:

- XIC, XIO, BST, LIM, EQU, NEQ, LES, GRT, LEQ, GEQ

En AuroraLD Studio las instrucciones de entrada siempre se colocan en el lado izquierdo y son las responsables de la lógica que se debe de cumplir para ejecutar las salidas. El algoritmo que se implementa evalúa cíclicamente de arriba hacia abajo línea a línea verificando si las condiciones de entrada se cumplen.

Primero se verifica aquellas instrucciones que se encuentran en Paralelo. Para ello se determina el estado actual de dichas instrucciones colocando su valor respectivo (“0” o “1”) tal como se muestra en la Figura 193.

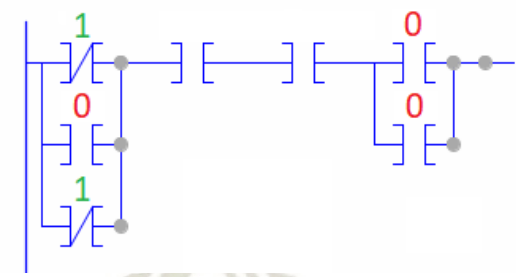


Figura 193: Determinando estados de instrucciones que están en paralelo

Se realiza una suma de los estados de un grupo obteniendo así un resultado final para ese grupo tal como se muestra en la Figura 194. Si hay más de un grupo de instrucciones que se encuentran en paralelo se realiza el procedimiento mencionado anteriormente.

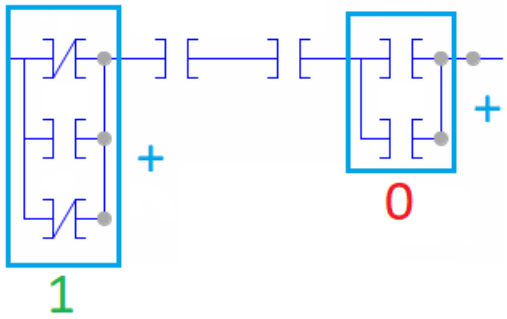


Figura 194: Suma de instrucciones de entrada que están en paralelo

Cuando finalmente se tiene un resultado de cada grupo de instrucciones en paralelo se multiplican. En el caso de la Figura 194, el resultado de la multiplicación de ambos grupos de instrucciones nos da un resultado de "0".

Finalmente, se asignan los estados de las instrucciones restantes y se realiza una multiplicación entre todos los estados incluyendo el obtenido en el procedimiento anterior. En la Figura 195 observamos que solo restan dos instrucciones XIC cuyo estado actual es "0" por lo tanto hay que multiplicar esos estados por el resultado que obtuvo de las instrucciones en paralelo teniendo finalmente un resultado de "0". Este resultado indica que las condiciones de entrada no se satisfacen y por lo tanto las salidas asociadas a dicha línea no se deben de ejecutar.

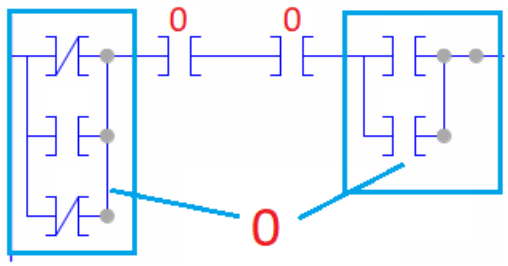


Figura 195: Multiplicación de estados de instrucciones de entrada

Todas las operaciones mencionadas son a nivel Lógico, es decir, que los únicos valores que se pueden obtener como respuesta son "0" o "1". Si al final de todas las

operaciones mencionadas se obtiene un valor de “1” entonces las salidas asociadas a esa línea se activan y si se obtiene un valor de “0” las salidas asociadas a la línea no se activan.

Todo lo detallado anteriormente respecto al algoritmo se muestra en forma de código en la Figura 196 para la instrucción XIC.

```
for (int k=0; k<contador; k++) {
  for (int j=0; j<bst_counter; j++) {
    if ((BST[j].belong==u)&&(BST[j].x==xx[k])) {

      if (one_time==0) {
        operation[tt]=true;
        for (int i=0; i<xic_counter; i++) {
          XIC[i].data();
          if ((XIC[i].belong==u)&&(XIC[i].x>=xx[k])&&(XIC[i].x<xarm[k])
            &&(XIC[i].y==RUNG[u].y)) {
            operation[tt]=operation[tt]&&XIC[i].state;
          }
        }
```

Figura 196: Algoritmo de cálculo de instrucción XIC en paralelo

3.11. Eliminar instrucciones Ladder

El algoritmo que permite eliminar instrucciones se basa en hacer los objetos creados Nulos (Null). Si se reduce el contador de instrucciones (de cada tipo) y sus atributos se hacen Nulos el objeto desaparece de la vista del usuario.

A medida que se agrega una nueva instrucción en el entorno Ladder aumenta el contador relacionado a dicha instrucción indicando así la cantidad de instrucciones de ese tipo. Cuando se elimina una instrucción todos los parámetros de las demás instrucciones de ese tipo son pasadas a un objeto previo y siempre se elimina el último objeto creado. Finalmente se reduce el contador de dicha instrucción en un uno.

Todo lo mencionado anteriormente se ejecuta siempre y cuando el objeto eliminado no sea el último, ya que en caso contrario no es necesario pasar ningún parámetro y basta con eliminar el último objeto creado.

En AuroraLD Studio se accede a los atributos de los objetos empleando el nombre del objeto, un índice (contador relacionado a una instrucción) el cual se coloca entre corchetes seguido por un punto y el nombre del atributo al que se desea acceder, por ejemplo: XIC[4].belong.

Se detalla el algoritmo para borrar instrucciones con el siguiente ejemplo:

Supongamos que tenemos 4 instrucciones del tipo XIC y se desea borrar la instrucción que se encuentra en la posición 2.

- El contador de la instrucción XIC será 3 puesto que el primer objeto creado es 0, luego 1 y así hasta 3.

$$XIC[0], XIC[1], XIC[2], XIC[3] \rightarrow 4\ Instrucciones$$

- La instrucción a eliminar hace referencia al contador de instrucción 1 ya que es la segunda instrucción XIC agregada (posición 2).
- Se deben pasar todos los atributos de las instrucciones XIC cuyo contador de instrucción sea mayor a la instrucción que se desea borrar, en este caso se empieza en 2.

$$XIC[0]\ \boldsymbol{XIC[1]}\ XIC[2]\ XIC[3]$$

- En un bucle se accede a los atributos de la instrucción que se desea borrar y se escribe en esa instrucción los parámetros de la siguiente instrucción. Se continúa haciendo esto hasta llegar a la última instrucción. En otras palabras, los atributos de la instrucción XIC[2] son pasados a XIC[1] y los parámetros de la instrucción XIC[3] son pasados a XIC[2].
- Finalmente se elimina la última instrucción.

Todo lo explicado en el ejemplo anterior se puede visualizar en la Figura 197. En esta figura se muestra el algoritmo implementado para una instrucción XIC. El resto de instrucciones posee un algoritmo idéntico en función de sus atributos.

```
for (int i=num+1; i<xic_counter; i++) {
  XIC[i-1].name=XIC[i].name;
  XIC[i-1].type=XIC[i].type;
  XIC[i-1].row_tag=XIC[i].row_tag;
  XIC[i-1].x=XIC[i].x;
  XIC[i-1].y=XIC[i].y;
  XIC[i-1].value=XIC[i].value;
  XIC[i-1].state=XIC[i].state;
  XIC[i-1].belong=XIC[i].belong;
  XIC[i-1].nn=70;
  XIC[i-1].exist=XIC[i].exist;
  XIC[i-1].exist2=XIC[i].exist2;
}
```

Figura 197: Algoritmo para eliminar instrucciones

3.12. Exportar proyecto a código Arduino

El algoritmo implementado para exportar el Diagrama Ladder a código entendible por la plataforma Arduino IDE se realizó tomando como base el código de Arduino, es decir, que en Processing se escribe el código de Arduino como Strings línea a línea considerando que algunas variables sean generales y se agreguen en función del diagrama Ladder.

En la Figura 198 se muestra parte del código que permite exportar las primeras líneas de un proyecto. En estas primeras líneas se tiene la versión del software utilizado, la placa que se emplea, la versión de la placa, hora, fecha y nombre de usuario que exporta el archivo. También se exporta variables cruciales para el funcionamiento del programa las cuales incluyen las entradas y salidas tanto analógicas como digitales.

```
String[] INIT = {
  "//AuroraLD Studio - " + verp,
  "//" + add_board_name,
  "//" + add_board_version,
  "//HOUR   "+hour()+":"+minute()+":"+second(),
  "//DATE   "+day()+"/"+month()+"/"+year(),
  "//"+ System.getProperty("user.name"),
  "",
  "//INPUTS AND OUTPUTS",
  digital_inputs,
  digital_outputs,
  analog_inputs,
  analog_outputs,
  "",
  "//SEND DATA",
  state_input,
  "",
  "//TIME TO SEND DATA",
  "unsigned long actual_time = 0;",
  "unsigned long previous_time = 0;",
  "",
  "//TEMPORARY VARIABLES",
  "int cont = 0;",
  "int num["+uio+"];",
```

Figura 198: Ejemplo de algoritmo implementado para exportar el programa a código Arduino

Finalmente, solo debe guardarse los Strings generados con una extensión ino de manera que se pueda abrir directamente con el software Arduino IDE. En la Figura 199 podemos observar como la función saveStrings permite guardar el programa de Arduino con la extensión ino. Además, una vez que se exporta el programa recibimos un mensaje de confirmación.

```
saveStrings(dir_name+"/"+prg_name+"_plc/"+prg_name+"_plc.ino", PRG_ALL);
showMessageDialog(frame, "Project "+prg_name+
  " has been exported as "+prg_name+"_plc.ino", "Export", INFORMATION_MESSAGE);
```

Figura 199: Código que permite exportar el programa a código Arduino

Una vez el código ha sido exportado ya se encuentra listo para ser cargado en el microcontrolador y ya no será necesario mantenerlo conectado a la computadora para que ejecute la lógica Ladder implementada. Es importante mencionar que una vez que el código se exporta se abrirá automáticamente siempre y cuando se tenga el software de Arduino instalado. En la Figura 200 tenemos un ejemplo de código exportado en la que se muestra las primeras líneas con información básica para el funcionamiento.

```
//AuroraLD Studio - 1.8 (08/04/2021)
//Arduino Nano
//Nano_AuroraLD
//HOUR   18:16:14
//DATE   23/4/2021
//damzm

//INPUTS AND OUTPUTS
int DI[6] = {2,3,4,5,7,8};
int DO[6] = {12,13,14,15,16,17};
byte AI[4] = {A4,A5,A6,A7};
int AO[4] = {6,9,10,11};

//SEND DATA
int STATE[6];

//TIME TO SEND DATA
unsigned long actual_time = 0;
unsigned long previous_time = 0;

//TEMPORARY VARIABLES
int cont = 0;
int num[100];
int DATA[100];

//VARIABLES (PROGRAM TAGS)
```

Figura 200: Ejemplo código exportado

3.13. Compilación del programa

La compilación del programa se ejecuta tanto a nivel del entorno Ladder como en los Program Tags. A nivel del entorno Ladder se verifica línea a línea que las instrucciones contengan todos los parámetros definidos y configurados, se busca repetición de tags en instrucciones que no son posibles de acuerdo a su funcionamiento y finalmente también se evalúa la existencia de líneas (Rungs) en blanco.

Este proceso de compilación es importante ya que ayuda a verificar que todas las variables que se necesitan para la ejecución existan, ya que en caso contrario generaría errores en la ejecución haciendo imposible correr el programa, deteniendo el software y/o teniendo resultados inesperados en la ejecución.

```
if (t_ladder==0) {
  boolean done=false;
  for (int i=0; i<=ptag_counter; i++) {
    for (int j=0; j<=ptag_counter; j++) {
      if (tag_name[i].equals(tag_name[j])) {
        if (i!=j) {
          showMessageDialog(frame, "Tag "+j+" already exists", "Repetitive tag", ERROR_MESSAGE);
          t_ladder=1;
          done=true;
          break;
        }
      }
    }
    if (done==true) {
      break;
    }
  }
}
```

Figura 201: Ejemplo de algoritmo que verifica la repetición de Tags

El algoritmo implementado está básicamente basado en bucles for que verifican la repetición de nombres y direcciones como se muestra en la Figura 201.

3.14. Guardar y Cargar proyectos a AuroraLD Studio

Los programas guardados en AuroraLD Studio se componen de varios elementos que se guardan en una carpeta con el nombre del proyecto tal como se muestra en la Figura 202. En la figura se muestra 4 archivos fundamentales que se generan de forma automática tras crear un nuevo proyecto.

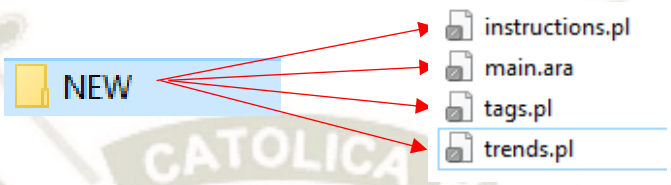


Figura 202: Folder de proyecto

- La carpeta lleva el nombre del proyecto.
- El archivo “main.ara” guarda el nombre del proyecto, tiempo y fecha. Es el archivo principal de todo proyecto en AuroraLD Studio. Cuando se cargue un proyecto este es el archivo que debe abrirse.
- El archivo “instructions.pl” guarda todas las instrucciones que se hayan colocado en el espacio de trabajo Ladder.
- El archivo “tags.pl” guarda todos los tags creados en el programa.
- Finalmente, el archivo “trends.pl” guarda toda la configuración referida al entorno de tendencias.

Se ha empleado el cuadro de diálogo JFileChooser de la librería Java Swing la cual permite seleccionar y ubicar las rutas o directorios donde se desean guardar los proyectos creados en AuroraLD Studio. Todo lo que se debe hacer en la ventana mostrada en la Figura 203 es colocar un nombre al proyecto y proceder a hacer click en Save.

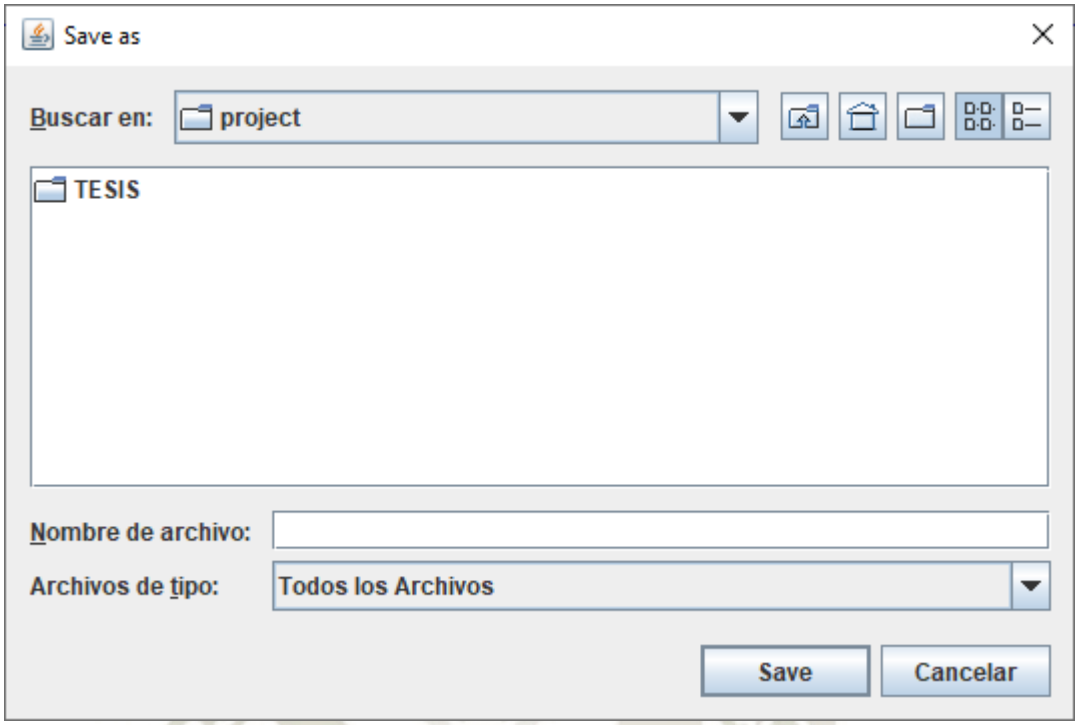


Figura 203: Ventana para guardar proyectos

Parte del algoritmo implementado para realizar el guardado se muestra en la Figura 204. En el código se invoca a la ventana de guardar como y posteriormente si se ha hecho click en Save se procede a guardar la ruta de la carpeta en una variable denominada "project_directory" y el nombre de la carpeta se almacena en "file_name". Siempre y cuando no exista un proyecto con el nombre escogido en la ruta especificada, el proyecto se guarda sin problema alguno, en caso contrario se muestra un error indicando al usuario que el proyecto ya existe. En AuroraLD Studio no se permite sobreescribir proyectos por lo que las carpetas deben tener siempre nombres distintos.

```
JFileChooser save_choose = new JFileChooser(dataPath(sketchPath("project/")));
save_choose.setDialogTitle("Save as");
//save_choose.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

String project_directory="";
String file_name="";
if (save_choose.showDialog(frame, "Save")==JFileChooser.APPROVE_OPTION) {
  project_directory=save_choose.getSelectedFile().getPath();
  file_name=save_choose.getSelectedFile().getName();
} else {
  project_directory=null;
  file_name=null;
}

if (file_name!=null) {
  project_directory=project_directory.substring(0, (project_directory.length()-
    file_name.length()));
  dir_name=project_directory+"/"+file_name;
  File fil = dataFile(project_directory+"/"+file_name+"/main.ara");

  if (fil.exists()==true) {
    println("EXISTE");
    showMessageDialog(frame, "Project already exists", "New", ERROR_MESSAGE);
```

Figura 204: Código ventana guardado

Dentro de las opciones de guardado se llama a una función llamada Save que básicamente almacena todos los parámetros correspondientes convirtiendo los datos a Strings para guardarlos como archivos de texto con la extensión correspondiente.

En el caso del archivo principal “main.ara” se ha implementado el código mostrado en la Figura 205. Se almacena el nombre del proyecto, fecha, hora, etc.

```
try {
  //MAIN DATA
  String file_data[];
  String[] project_name = {"//PROJECT NAME", prg_name};
  String[] project_time = {"//TIME AND DATE", hour()+":"+minute()+":"+second(),
    day()+"/"+month()+"/"+year()};
  String[] rail_config = {"//RAIL", str(rail_y)};
  file_data = concat(project_name, concat(project_time, rail_config));
  saveStrings(dir_name+"/main.ara", file_data);
}
catch(Exception e) {
  invalid_name = true;
}
```

Figura 205: Código main.ara

Las instrucciones Ladder se deben guardar en un orden específico previamente establecido de manera que todos los atributos de los objetos creados puedan ser cargados en el mismo orden y no presentar problemas.

Primero, se guardan datos referentes a los contadores de cada instrucción tal cual se muestra en la Figura 206. Esto se hace con la finalidad de saber cuántas instrucciones de un determinado tipo se han agregado a un proyecto.

```
String[] instruction_counter={str(rung_counter), str(xic_counter), str(xio_counter),
  str(bst_counter), str(ote_counter), str(otl_counter), str(otu_counter),
  str(ton_counter), str(tof_counter), str(rto_counter), str(ctu_counter),
  str(ctd_counter), str(res_counter), str(lim_counter), str(equ_counter),
  str(neq_counter), str(les_counter), str(grt_counter), str(leq_counter),
  str(geq_counter), str(add_counter), str(sub_counter), str(mul_counter),
  str(div_counter), str(sqr_counter), str(neg_counter), str(mod_counter),
  str(abs_counter), str(cpt_counter), str(scp_counter), str(mov_counter),
  str(acs_counter), str(asn_counter), str(atn_counter), str(cos_counter),
  str(sin_counter), str(tan_counter), str(deg_counter), str(rad_counter),
  str(pid_counter)};
```

Figura 206: Contadores de instrucciones

Segundo, se guardan todos los atributos de las instrucciones en un orden específico empleando para ello bucles for. En la Figura 207 se muestra el código que permite guardar los atributos de la instrucción XIC. Las demás instrucciones tienen un algoritmo similar.

```
String xic_param[] = new String[xic_counter*12];
contador = 0;
for (int i=0; i<xic_counter; i++) {
  xic_param[contador]=XIC[i].name;
  contador++;
  xic_param[contador]=XIC[i].type;
  contador++;
  xic_param[contador]=str(XIC[i].num);
  contador++;
  xic_param[contador]=str(XIC[i].x);
  contador++;
  xic_param[contador]=str(XIC[i].y);
  contador++;
  xic_param[contador]=str(XIC[i].value);
  contador++;
  xic_param[contador]=str(XIC[i].state);
  contador++;
  xic_param[contador]=str(XIC[i].belong);
  contador++;
  xic_param[contador]=str(XIC[i].row_tag);
  contador++;
  xic_param[contador]=str(XIC[i].nn);
  contador++;
  xic_param[contador]=str(XIC[i].exist);
  contador++;
  xic_param[contador]=str(XIC[i].exist2);
  contador++;
}
```

Figura 207: Código que guarda los atributos de la instrucción XIC

Finalmente, se usa la función saveStrings para almacenar todos los Strings en un archivo llamado “instructions.pl” dentro de la carpeta de proyecto correspondiente. En la Figura 208 se muestra el código que encadena todos los strings que contienen los atributos necesarios e importantes de cada instrucción y posteriormente se guarda en el archivo especificado.

```
file_instructions=concat(instruction_counter, concat(rung_param,
  concat(xic_param, concat(xio_param, concat(bst_param, concat(ote_param,
  concat(otl_param, concat(otu_param, concat(ton_param, concat(tof_param,
  concat(rto_param, concat(ctu_param, concat(ctd_param, concat(res_param,
  concat(lim_param, concat(equ_param, concat(neq_param, concat(les_param,
  concat(grt_param, concat(leq_param, concat(geq_param, concat(add_param,
  concat(sub_param, concat(mul_param, concat(div_param, concat(sqr_param,
  concat(neg_param, concat(mod_param, concat(abs_param, concat(cpt_param,
  concat(scp_param, concat(mov_param, concat(acs_param, concat(asn_param,
  concat(atn_param, concat(cos_param, concat(sin_param, concat(tan_param,
  concat(deg_param, concat(rad_param, pid_param))))))))))))))))))))))))));

saveStrings(dir_name+"/instructions.pl", file_instructions);
```

Figura 208: Guardado instructions.pl

Por otro lado, al cargar un proyecto en AuroraLD Studio se procede a leer los Strings de los archivos que se están cargando. Para ello se hace uso de la función loadStrings que permite cargar los datos guardados línea a línea en un array denominado “main_file” tal como se muestra en la Figura 209.

```
//MAIN
String project = project_directory+"main.ara";
String[] main_file = loadStrings(project);
prg_name=main_file[1];
surface.setTitle("AuroraLD Studio - "+prg_name);
```

Figura 209: Código para cargar main.ara

El archivo que se carga en AuroraLD Studio es “main.ara” y se escoge el archivo a través de un JFileChooser obtenido así la ruta de la carpeta de proyecto y acceso al resto de archivos.

La ventaja de guardar los datos en un orden específico es que al momento de cargar un archivo ya se conoce la posición de cada variable por lo que se puede volver a asignar los valores guardados a cada una de las variables correspondientes. En la Figura 210 se muestra el código que permite asignar los contadores de cada instrucción a su respectiva variable.

```
//INSTRUCTIONS
////////////////////////////////
String project_2 = project_directory+"instructions.pl";
String[] instruction_file = loadStrings(project_2);
//println(instruction_file);
//NUMBER OF EACH INSTRUCTION
rung_counter=int(instruction_file[0]);
xic_counter=int(instruction_file[1]);
xio_counter=int(instruction_file[2]);
bst_counter=int(instruction_file[3]);
ote_counter=int(instruction_file[4]);
otl_counter=int(instruction_file[5]);
```

Figura 210: Carga de contadores de instrucciones

En las instrucciones se emplea el constructor de la clase para volver a generar los objetos con los atributos guardados. Para ello el código de forma iterativa en función del contador de instrucción asigna los atributos y vuelve a crear los objetos tal como se muestra en la Figura 211. El código presentado en esta figura hace referencia la instrucción CTU pero funciona de forma similar para el resto de instrucciones.

```
//CREATE CTU
for (int i=0; i<ctu_counter; i++) {
  contador++; //146
  CTU[i] = new COUNTER_UP(instruction_file[contador], instruction_file[contador+1],
    int(instruction_file[contador+2]), int(instruction_file[contador+3]),
    float(instruction_file[contador+4]), boolean(instruction_file[contador+5]),
    int(instruction_file[contador+6]), int(instruction_file[contador+7]),
    int(instruction_file[contador+8]), boolean(instruction_file[contador+9]),
    boolean(instruction_file[contador+10]));
  contador=contador+11; //157
  CTU[i].row_tag=int(instruction_file[contador]); //157
}
```

Figura 211: Ejemplo de carga instrucción CTU

3.15. Comunicación serial con Arduino

Para comunicar Arduino con el software es necesario cargar un programa previo que contiene el código necesario para el envío de datos al software, así como también un código único de identificación para que el software lo reconozca como un dispositivo compatible.

Una vez que se carga el programa en el microcontrolador se procede a realizar la conexión con la PC a través de un cable USB. Se puede verificar el puerto Serial de Comunicación en el administrador de dispositivos para estar seguros de cual escoger en AuroraLD Studio. En la Figura 212 observamos que el hardware ha sido conectado en el puerto de comunicación serial 5 por lo que debemos escoger en AuroraLD Studio aquel que haga referencia a dicho puerto (COM5).

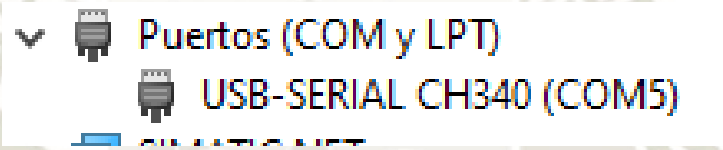


Figura 212: Puerto Serial PLC1805 v1.0

La comunicación con el microcontrolador se realiza de forma bidireccional empleando el puerto Serial. El tipo de comunicación es UART también conocido como Transmisor-Receptor Asíncrono Universal y se realiza a 115200 baudios. La selección de la velocidad de transmisión se realizó mediante prueba y error ya que existe un delay al momento de enviar y recibir datos. De acuerdo a las pruebas realizadas se obtuvieron mejores resultados empleando la máxima velocidad de transmisión de datos disponible por las placas Arduino UNO y NANO.

Desde el software se envían datos en una trama de información la cual se muestra en la Figura 213. La trama contiene datos acerca de las variables digitales y analógicas definidas en el software de acuerdo al tipo de microcontrolador seleccionado.

0	1	1	0	1	0	0	186	120	255
Salida Digital						Salida Analógica			

Figura 213: Trama de datos de salida del software

Los primeros bits hacen referencia a la salida digital de cada una de las direcciones externas definidas para los microcontroladores Arduino NANO y Arduino UNO, estos valores son calculados en AuroraLD Studio. Al finalizar la ejecución de una línea se envían datos al microcontrolador cambiando así los estados de dichas salidas digitales. De manera similar se procede con las variables analógicas. Para otras placas el esquema es similar ya que en los primeros bits se tiene información acerca de las salidas digitales seguido por las salidas analógicas.

Los datos contenidos en la trama de datos son separados por comas y enviados a través del puerto serial empleando la función mostrada en la Figura 214.

```
PLC.write(q);
```

Figura 214: Función write para enviar datos

Por otro lado, desde Arduino se envía información real de los actuadores y/o sensores conectados al PLC en la trama de datos mostrada en la Figura 215. Los datos recibidos se almacenan en la tabla de datos del programa para ser utilizados por AuroraLD Studio.

Los primeros bits hacen referencia a los sensores digitales que se han colocado en el software. Los siguientes datos hacen referencia a los sensores analógicos, estos valores analógicos pueden tomar como máximo valores de 1023. Este valor máximo está definido por la resolución del conversor analógico digital que poseen las placas Arduino Uno y Arduino Nano.

0	1	1	0	1	0	0	186	120	255	1607
Entrada Digital						Entrada Analógica				Código

Figura 215: Trama de datos de entrada al software

El último dato de la trama de salida del PLC se denomina código y es un valor que permite identificar y diferenciar las distintas placas de trabajo. Adicionalmente permite establecer una comunicación con el software y verificar si se ha interrumpido la comunicación con el PLC ya que el dato de código se verifica constantemente. Si el código se deja de recibir, el software asume que el dispositivo ha sido desconectado e informa al usuario. La verificación de la comunicación del hardware con el software se muestra en la Figura 216. Se observa en la figura que mediante una condición if se comprueba que el último dato recibo sea el código 5518 verificando así la comunicación del hardware con el software.

```
//PLC1805
if((select_board==0)&&(input_dataUno[10]==5518)){
sth_connected=false;
connected = true;
showMessageDialog(frame,"          Device connected");
}
```

Figura 216: Prueba de conexión en base al código recibido

La trama de datos de entrada es recogida por el software y asignada a una serie de variables dentro de la Tabla de Datos del Programa. Las variables a las que se asigna no son visibles hasta no crear los Tags respectivos, pero se asocian directamente a las direcciones especificadas en el software correspondientes a las direcciones físicas del PLC.

```
String DI[] = split(myString, ',');

if(DI.length>1)
{
  for(int i=0;i<(number_DI+number_AI+1);i++){
  //value1 = int(DI[0]);
  //value2 = int(DI[1]); //remember to create another value variable
 if((select_board==0)||(select_board==20)){
 input_dataUno[i]=int(DI[i]);
 }
 else if(select_board==40){
 input_anyBoard[i]=int(DI[i]);
 }
```

Figura 217: Recepción de datos en AuroraLD Studio

En la Figura 217 mediante un bucle for se comprueba todos los datos que provengan del microcontrolador y se almacenan en las variables input_dataUno en el caso de un Arduino Nano o Arduino Uno.

Como se especificó antes, el enlace de comunicación con el PLC es monitoreado constantemente por si surge alguna desconexión alertando al usuario en caso el hardware se desconecte. Incluso es posible emplear bluetooth para realizar la comunicación serial, pero presenta fallos si el dispositivo se desconecta impidiendo el funcionamiento correcto y obligando al usuario a reiniciar el software. Bajo otras condiciones que no impliquen una desconexión del hardware, la comunicación por bluetooth permite utilizar todas las funciones del software.

3.16. Programa para comunicar Arduino con AuroraLD Studio

El microcontrolador debe tener un código previamente cargado para poder trabajar con AuroraLD Studio. Este código contiene lo necesario para enviar y recibir los valores digitales y/o analógicos. En AuroraLD Studio se conoce a este código base como Controller. En la Figura 218 se muestra el código base que permite leer los estados de las entradas

digitales y analógicas, así como también enviar los datos mediante “serial.print” al software. Los datos son enviados a AuroraLD Studio cada 20 milisegundos.

```
for(int i=0;i<6;i++){
if(DATA[i] == 1){
digitalWrite(DO[i], HIGH);
}
else if (DATA[i] == 0){
digitalWrite(DO[i], LOW);
}
}

for(int i = 6; i < 10; i++){
analogWrite(AO[i-6], DATA[i]);
}

if(actual_time-previous_time>=20){
previous_time=actual_time;
for(int i = 0; i < 10; i++){
if(i < 6){
Serial.print(STATE[i]);
Serial.print(",");
}else{
Serial.print(analogRead(AI[i-6]));
Serial.print(",");
}
}
Serial.println(5518);
}
}
```

Figura 218: Recepción de datos digitales y envío de valores analógicos

En las opciones de comunicación se puede acceder a la opción Generate Controller tal cual se muestra en la Figura 219. La opción Generate Controller generará de forma automática el programa base mostrado previamente en la Figura 218. Como ya se indico en otras ocasiones, este programa se debe cargar en el microcontrolador para poder establecer una comunicación con AuroraLD Studio.

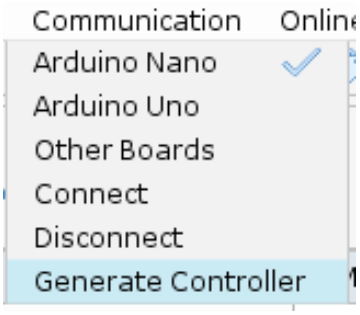


Figura 219: Opción Generate Controller

3.17. Ciclo de Trabajo (Diagrama Ladder)

El ciclo de trabajo que se define a continuación se ha determinado de forma experimental empleando la consola de processing y el comando millis() para determinar el tiempo que transcurre en determinadas parte del software.

La ejecución básica de operación del diagrama Ladder en AuroraLD Studio se puede dividir en 3 fases:

- Lectura de señales desde la interfaz de entradas
- Procesamiento de la información con la finalidad de ejecutar la lógica del programa
- Escritura de señales en la interfaz de salidas

En vista de lo mencionado anteriormente podemos decir que el ciclo de trabajo expresado en tiempo está directamente vinculado con los siguientes procesos:

- El tiempo que le toma al programa recibir la información de las entradas desde la placa de desarrollo (Dependiendo del número de entradas).
- El tiempo que le toma al software ejecutar la lógica Ladder (Esto depende también de la cantidad de líneas que posea el programa, es decir, la longitud del programa).
- El tiempo que le toma al software enviar las salidas hacia la placa de desarrollo que se tenga conectada (Dependiendo del número de salidas).
- Tiempo que se demora en procesar las entradas y salidas el hardware conectado.

El tiempo que se toma el hardware en procesar la información de las entradas digitales y analógicas para ser enviadas a AuroraLD Studio se da en un intervalo de cada **20 milisegundos** (Parámetro que ha sido colocado a propósito en Arduino para evitar enviar tanta información de forma seguida en poco tiempo). Por otro lado, el tiempo que le toma al hardware procesar toda la información para las placas que soporta por defecto (Arduino Nano y Arduino Uno) es de aproximadamente **4 ciclos para cada milisegundo**.

El tiempo que le toma al software recibir la información del hardware es de aproximadamente **1 milisegundo,** tiempo que fue determinado usando el comando millis().

El tiempo que le toma al software procesar la lógica Ladder a ejecutar por línea va a depender de la complejidad de la lógica, siendo así en los casos donde se manejan múltiples instrucciones o variables el tiempo mayor y aproximado de **8 milisegundos** y en el caso de líneas simples (instrucciones que se encuentran en la pestaña "Common") el tiempo fluctúa entre **menor a 1 milisegundo, 1 milisegundo y 2 milisegundos**.

Finalmente, el tiempo que le toma al software juntar los datos de salidas y separarlos por comas es de aproximadamente **2 milisegundos** para los datos de salida del hardware que AuroraLD Studio soporta por defecto (Arduino Nano y Arduino Uno).

Haciendo una suma de los datos que tenemos respecto a la ejecución de la lógica Ladder podemos estimar que el software se tarde aproximadamente **5 milisegundos** para ejecutar una sola línea simple. De este dato hay que tener en cuenta que **1 milisegundo** es fijo para leer los datos provenientes del hardware y además **2 milisegundos** es un parámetro fijo para enviar los datos hacia el hardware. El resto del tiempo debe sumarse en función dela lógica que se implementa.

3.18. Entorno de tendencias

El entorno de Trends permite graficar, monitorear y ver la evolución de las variables de un determinado proceso. Se pueden graficar hasta un máximo de 9 variables tanto digitales como analógicas. Es posible también cambiar el tiempo de muestreo desde un valor mínimo de 50ms hasta un valor máximo no definido. El tiempo de muestreo depende mucha de la aplicación y de la cantidad de datos que se desea obtener de una determinada señal. En la Figura 220 se muestra la interfaz que se ha diseñado para el entorno de tendencias.

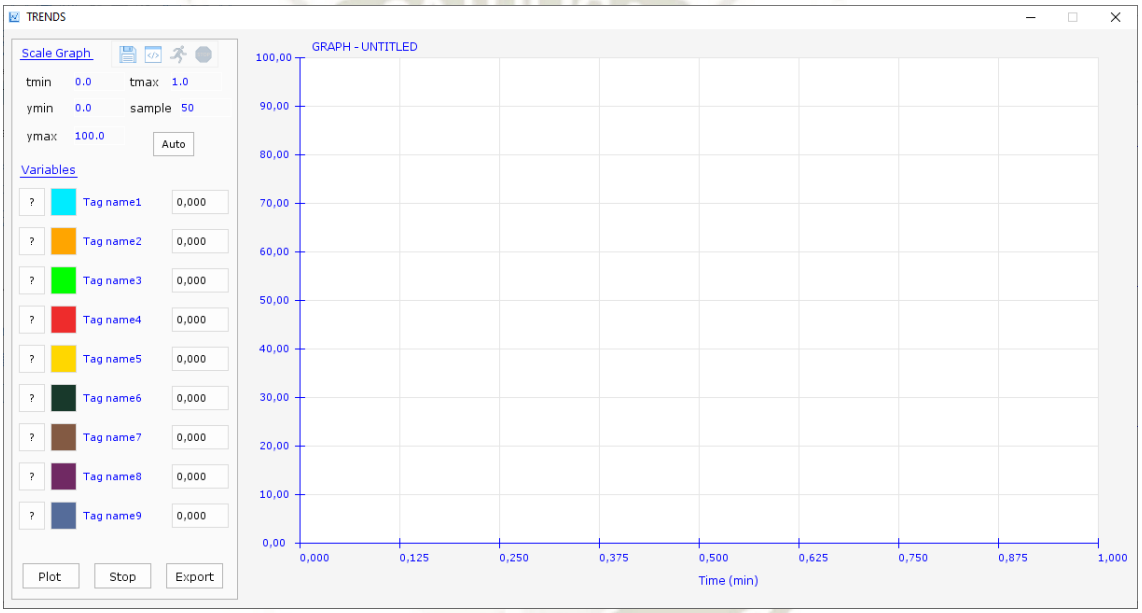


Figura 220: Entorno Trends

Los parámetros que se pueden variar se muestran en la Figura 221. Las opciones presentadas en la figura sirven para escalar la gráfica permitiendo así mostrar la gráfica entre un valor mínimo y máximo tanto en X como en Y. Algunos iconos que ya se explicaron en el entorno Ladder también se aprecian en la figura permitiendo así al usuario guardar, compilar e incluso controlar la ejecución de la lógica Ladder desde el entorno de tendencias.

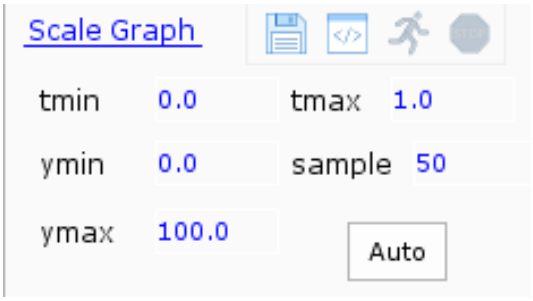


Figura 221: Parámetros Trends

La implementación de esta ventana de Trends se realiza creando una ventana adicional en el software definiendo así variables necesarias para almacenar los datos. Se considera un número de muestras en total de 120000, una vez que se hayan logrado almacenar la cantidad máxima de muestras especificadas se reinicia e inicia otra vez el muestreo de las señales a ser monitoreadas.

```
WINDOW5 TRENDS;
ControlP5 trend_graph;
Textfield time_max_graph;
Textfield time_min_graph;
Textfield ymin_graph;
Textfield ymax_graph;
Textfield sample_time_graph;

Table trend_table;

//WINDOW
boolean trend_front = false;

//SAMPLE
int num_sample=120000;

//VARIABLE TEST
float yt[] = new float[num_sample];
float xt[] = new float[num_sample];
float aa[] = new float[num_sample];
float bb[] = new float[num_sample];
```

Figura 222: Variables empleadas por el entorno Trends

Tanto los datos de la señal como los tiempos se almacenan en variables de tipo float, las cuales han sido declaradas como tipo array teniendo un máximo tamaño de acuerdo al número de datos de muestreo máximo tal como se muestra en la Figura 222.

Una vez que se tenga las gráficas deseadas estas se pueden exportar a Excel para el posterior tratamiento de los datos en caso sea necesario. Estos datos se exportan en un orden especifico exportando así las nueve variables teniendo valores de cero para aquellas variables que no han sido asociadas con ningún tag.

```
trend_table.addColumn("Time (ms)");
trend_table.addColumn(graph_1);
trend_table.addColumn(graph_2);
trend_table.addColumn(graph_3);
trend_table.addColumn(graph_4);
trend_table.addColumn(graph_5);
trend_table.addColumn(graph_6);
trend_table.addColumn(graph_7);
trend_table.addColumn(graph_8);
trend_table.addColumn(graph_9);
```

Figura 223: Exportar datos de Trends a Excel

Se exportan los datos a Excel empleando las funciones addColumn que permiten almacenar los valores del tipo y los valores respectivos de cada señal en una columna distinta, parte de este código se muestra en la Figura 223. El archivo exportado lleva por formato CSV y los datos se encuentran separados por comas. Conocer esta información es importante ya que ayuda al momento que se desee importar los datos en Excel.

3.19. Elementos SCADA

El entorno SCADA se compone básicamente de una ventana cuyas opciones se encuentran en la parte superior. Estas opciones en el entorno SCADA se conocen como elementos y son todos los gráficos con los que cuenta el entorno SCADA para realizar la

supervisión y/o control de un proceso. El entorno SCADA implementado se muestra en la Figura 224.

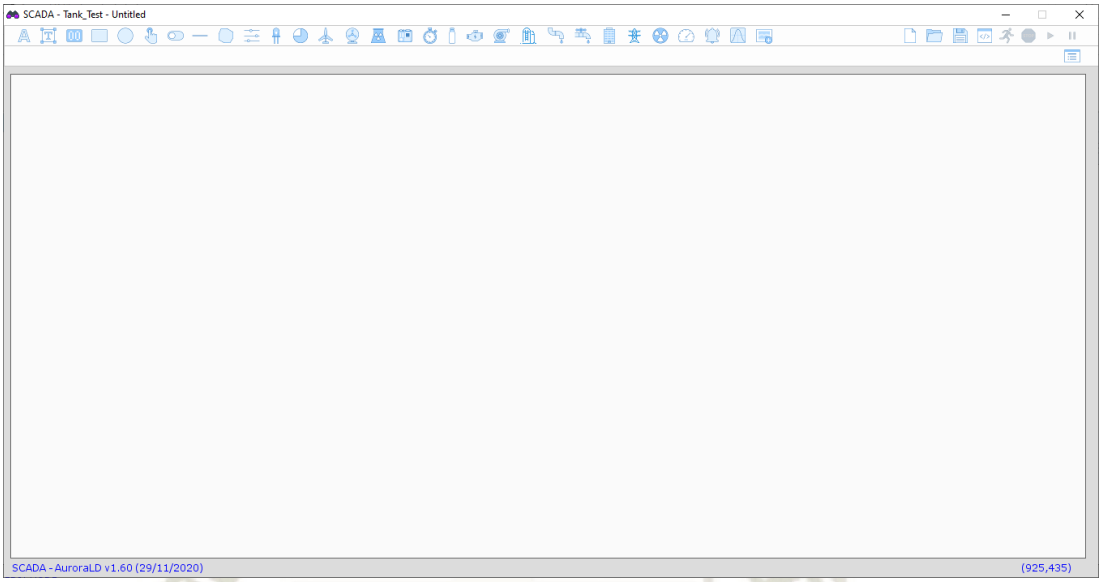


Figura 224: Entorno SCADA

AuroraLD Studio cuenta con 30 elementos diferentes los cuales se pueden observar en forma de íconos en la parte superior derecha del entorno SCADA tal como se muestra en la Figura 225. Para colocar estos elementos en el entorno SCADA es necesario seleccionar el ícono correspondiente y arrastrarlos al área de trabajo SCADA.

Figura 225: Elementos SCADA

Todos los elementos SCADA incluyen un parámetro denominado Layer el cual hace referencia a la capa en la que se dibuja un elemento en el entorno de trabajo SCADA. Cualquier elemento SCADA que se agregue al área de trabajo siempre se dibuja encima del resto de elementos. Es posible alterar el orden y cambiar un elemento determinado a otra capa a través de las opciones “bring to front” o “send to back”. Con las opciones

mencionadas se puede enviar un elemento al frente de todos los demás o al fondo del resto de elementos.

3.19.1. Texto (Text)

Este elemento permite agregar texto al entorno SCADA permitiendo así colocar títulos, nombre a componentes, etiquetas, etc. Un ejemplo de texto se muestra en la Figura 226.

Figura 226: Elemento Texto

Los elementos de Texto en AuroraLD Studio han sido definidos como una clase denominada HMI_TEXT a partir de la cual se generan los distintos objetos que se requieran colocar en el entorno SCADA. Los atributos de la clase HMI_TEXT se muestran en la Figura 227.

```
public class HMI_TEXT {

  final PApplet p;

  int x;
  int y;
  color col;
  int size;
  String texto;
  int layer;
  int num;
```

Figura 227: Atributos principales de la clase HMI_TEXT

- **x:** Posición X en pixeles del elemento texto.
- **y:** Posición Y en pixeles del elemento texto.
- **col:** Color RGB del texto.

- **size:** Tamaño en pixeles del texto.
- **texto:** Contenido del texto. Variable tipo String.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador de un determinado elemento.

3.19.2. Entrada texto (Input)

Este elemento permite a los usuarios crear una casilla de texto en la cual se pueden ingresar valores tanto enteros como decimales. Se asocia directamente a un tag al cual se le puede asignar el valor agregado en la casilla de texto. El elemento entrada de texto que se ha implementado en AuroraLD Studio se muestra en la Figura 228.

Figura 228: Elemento Entrada de texto

Los elementos de entrada de texto se implementan en base a una clase denominada HMI_IN cuyos atributos principales se detallan a continuación:

- **x:** Posición X en pixeles del elemento.
- **y:** Posición Y en pixeles del elemento.
- **col_txt:** Color del texto que se presenta en el elemento.
- **col_bg:** Color del fondo (background) que se presenta en el elemento.
- **size:** Tamaño del texto.
- **txt:** Texto mostrado en el elemento.
- **layer:** Número de capa en la que se dibuja un elemento.
- **num:** Identificador de un determinado elemento.
- **wr:** Ancho del elemento en pixeles.

- **hr:** Altura del elemento en pixeles.
- **tag:** Tag asociado al elemento.
- **row_tag:** Número de fila del tag que ocupa en la tabla de datos del programa.

En la Figura 229 se muestran los atributos previamente descritos como parte del código de la clase HMI_IN.

```
public class HMI_IN {

  final PApplet p;
  int x;
  int y;
  color col_txt;
  color col_bg;
  float size;
  String txt;
  int layer;
  int num;
  float wr;
  float hr;
  String tag;
  int row_tag = -1;
```

Figura 229: Atributos principales de la clase HMI_IN

3.19.3. Salida texto (Output)

El elemento de salida de texto permite mostrar valores numéricos y a diferencia del elemento de entrada de texto este elemento no permite modificar su valor. Permite al usuario mostrar valores de temperatura, valores escalados, valores calculados en la ejecución Ladder, etc. El elemento implementado se muestra en la Figura 230.

Figura 230: Elemento Salida de teto

Se implementa en base a una clase denominada HMI_OUT cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_txt:** Color RGB del texto.
- **col_rect:** Color RGB del rectángulo.
- **size:** Tamaño del texto en pixeles.
- **layer:** Número de capa en el cual se dibuja el elemento.
- **num:** Identificador único del elemento.
- **wr:** Ancho del elemento en pixeles.
- **hr:** Alto del elemento en pixeles.
- **obox_decimal:** Cantidad de dígitos de la parte decimal que se desea mostrar.
- **obox_int:** Cantidad de dígitos de la parte entera que se desea mostrar.
- **tag:** Tag asociado al elemento cuyo valor se mostrará.
- **row_tag:** Número de fila donde se encuentra ubicado el tag en la tabla de datos del programa.

El código que permite declarar los atributos dentro de la clase HMI_OUT implementada se muestra en la Figura 231.

```
public class HMI_OUT {

  final PApplet p;
  int x;
  int y;
  color col_txt;
  color col_rect;
  float size;
  int layer;
  int num;
  float wr;
  float hr;
  int obox_decimal;
  int obox_int;
  String tag;
  int row_tag = -1;
```

Figura 231: Atributos principales de la clase HMI_OUT

3.19.4. Rectángulo (Rectangle)

El elemento rectángulo permite dibujar un rectángulo con valores de ancho y alto especificados por el usuario, así como modificar el color de la figura. En la Figura 232 se muestra el elemento rectángulo implementado en AuroraLD Studio.

Figura 232: Elemento Rectángulo

Este elemento se implementa en base a una clase denominada HMI_RECT, la cual contiene los siguientes atributos:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.

- **col_rect:** Color RGB del rectángulo.
- **rect_stroke:** Color RGB del contorno del rectángulo.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador del elemento. Valor único e irrepetible que incrementa conforme se agregan más elementos rectángulos.
- **wr:** Ancho del elemento en pixeles.
- **hr:** Alto del elemento en pixeles.

```
public class HMI_RECT {

  final PApplet p;
  int x;
  int y;
  color col_rect;
  color rect_stroke;
  int layer;
  int num;
  float wr;
  float hr;
```

Figura 233: Atributos principales de la clase HMI_RECT

3.19.5. Círculo (Circle)

Este elemento permite crear un círculo por defecto, sin embargo, entre sus parámetros es posible modificar el ancho y alto. Si las proporciones se mantienen siempre se tendrá una circunferencia, pero si el ancho y alto son distintos se tendrá una elipse. El elemento implementado en AuroraLD Studio se muestra en la Figura 234.

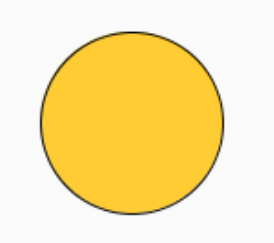


Figura 234: Elemento Círculo

Se implementa este elemento en base a una clase denominada HMI_CIRC cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_circ:** Color RGB del elemento.
- **circ_stroke:** Color RGB del contorno del elemento.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador del elemento.
- **wr:** Ancho del elemento en pixeles.
- **hr:** Alto del elemento en pixeles.

Los atributos detallados para el elemento circulo se muestran en la Figura 235.

```
public class HMI_CIRC {

  final PApplet p;
  int x;
  int y;
  color col_circ;
  color circ_stroke;
  int layer;
  int num;
  float wr;
  float hr;
```

Figura 235: Atributos principales de la clase HMI_CIRC

3.19.6. Botón (Button)

El elemento botón funciona solamente con tags del tipo digital y sirve para cambiar estados de falso a verdadero o viceversa enviando para esto 0 o 1 al tag asociado.

El valor del tag pasa a 1 mientras se mantiene pulsado el botón y regresa a 0 si se suelta. Los datos se escriben directamente en la columna de valor en la tabla de datos del programa siempre y cuando sean tags con dirección internal.

Figura 236: Elemento Botón

Este elemento se implementa en base a la clase denominada HMI_BUTTON cuyas variables principales se detallan a continuación:

- **x:** Posición X en pixeles del elemento.
- **y:** Posición Y en pixeles del elemento.
- **col_txt:** Color RGB del texto.
- **col_button:** Color RGB del botón.
- **texto:** String que almacena la etiqueta del botón.
- **layer:** Número de capa en el que ha sido dibujado el elemento.
- **num:** Identificador único del elemento.
- **tag:** Variable tipo string que guarda el tag asociado con el elemento.
- **row_tag:** Número de fila en el que se encuentra el tag en la tabla de datos del programa.

```
public class HMI_BUTTON {

  final PApplet p;
  int x;
  int y;
  color col_txt;
  color col_button;
  String texto;
  int layer;
  int num;
  String tag;
  int row_tag = -1;
```

Figura 237: Atributos principales de la clase HMI_BUTTON

3.19.7. Interruptor (Switch)

El funcionamiento del elemento interruptor es similar al elemento botón con la única diferencia de que se almacena el estado al que se cambia el tag, es decir, que no se requiere

tener pulsado el elemento para mantener el cambio de estado. En AuroraLD Studio se han implementado 3 tipos diferentes de switch los cuales se muestran en la Figura 238.

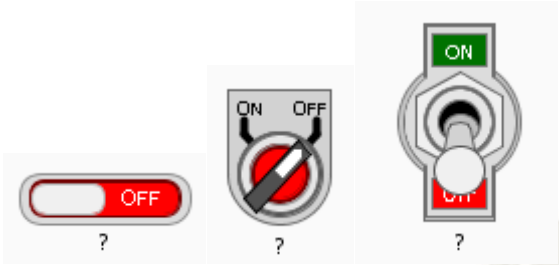


Figura 238: Elemento Interruptor

Este elemento se implementa en base a una clase denominada HMI_SWITCH cuyos atributos principales se detallan a continuación:

- **x:** Posición X en pixeles del elemento.
- **y:** Posición Y en pixeles del elemento.
- **col_switch_on:** Color RGB del elemento cuando se encuentra activo.
- **col_switch_off:** Color RGB del elemento cuando se encuentra inactivo.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **switch_type:** Existen tres tipos de interruptores que se pueden escoger de acuerdo a lo mostrado en la Figura 238.
- **tag:** Variable tipo string que guarda el tag asociado al elemento.
- **row_tag:** Fila en la que se encuentra el tag en la tabla de datos del programa.

```
public class HMI_SWITCH

  final PApplet p;
  int x;
  int y;
  color col_switch_on;
  color col_switch_off;
  int layer;
  int num;
  int switch_type; //3 t
  String tag;
  int row_tag = -1;
```

Figura 239: Atributos principales de la clase HMI_SWITCH

3.19.8. Línea (Line)

El elemento línea permite dibujar una línea continua según requiera el usuario. Este elemento no se relaciona con ningún tag puesto que en sus propiedades los únicos parámetros a modificar son el color y el grosor de la línea.

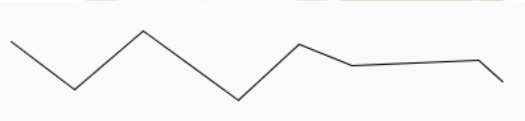


Figura 240: Elemento Línea

La implementación de este elemento se realiza en base a la clase HMI_LINE cuyos atributos principales se detallan a continuación:

- **x:** Vector que almacena como máximo 200 puntos correspondientes a la coordenada X en pixeles.
- **y:** Vector que almacena como máximo 200 puntos correspondientes a la coordenada Y en pixeles.

- **vx_num:** Número de vértices.
- **x1:** Coordenada X en pixeles del punto inicial.
- **y1:** Coordenada Y en pixeles del punto inicial.
- **col_line:** Color RGB de la línea.
- **line_stroke:** Grosor de la línea en pixeles.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.

```
public class HMI_LINE {

  final PApplet p;
  int[] x = new int[200];
  int[] y = new int[200];
  int vx_num = 0;
  int x1;
  int y1;
  color col_line;
  float line_stroke;
  int layer;
  int num;
```

Figura 241: Atributos principales de la clase HMI_LINE

3.19.9. Polígono (Polygon)

El elemento polígono permite al usuario crear cualquier figura considerando un máximo de 200 vértices. Las figuras creadas no pueden ser modificadas y si se requiere realizar algún cambio será necesario volver a crear otra figura. Este elemento no realiza ninguna acción y por ende no se asocia con ningún Tag, sin embargo, se pueden agregar animaciones a este elemento.

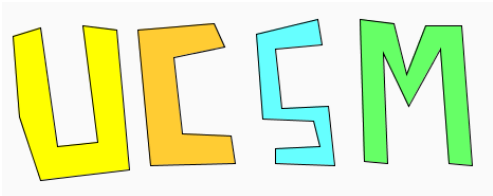


Figura 242: Elemento Polígono

Se implementa este elemento en base a la clase HMI_POLYGON cuyos atributos principales se detallan a continuación:

- **x:** Vector que almacena como máximo 200 coordenadas X en pixeles del elemento.
- **y:** Vector que almacena como máximo 200 coordenadas Y en pixeles del elemento.
- **vx_num:** Número de vértices.
- **col_pol:** Color RGB del elemento.
- **pol_stroke:** Color RGB del contorno del elemento.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.

```
public class HMI_POLYGON {

  final PApplet p;
  int[] x = new int[200];
  int[] y = new int[200];
  int vx_num = 0;
  color col_pol;
  color pol_stroke;
  int layer;
  int num;
```

Figura 243: Atributos principales de la clase HMI_POLYGON

3.19.10. Slider

El elemento slider se asocia a un tag de tipo analógico y solamente si ha sido declarado como un tag de tipo internal. Permite variar el valor del tag asociado en un rango especificado por el usuario.

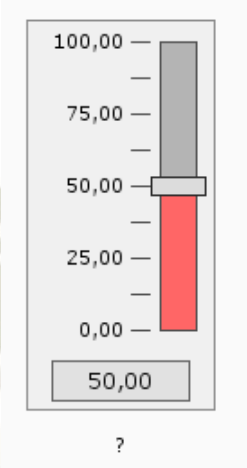


Figura 244: Elemento Slider

Se implementa este elemento en base a la clase HMI_SLIDER cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **min_s:** Mínimo valor que puede asumir el tag asociado.
- **max_s:** Máximo valor que puede asumir el tag asociado.
- **slider_result:** Variable donde se almacena el valor actual del slider en función de la posición en pixeles y el rango mínimo y máximo especificado.
- **fill_slider:** Color RGB del slider.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **tag:** Variable tipo string que almacena el tag asociado.

- **row_tag:** Número de fila en la que se ubica el tag en la tabla de datos del programa.

```
public class HMI_SLIDER {

  final PApplet p;
  int x;
  int y;
  float min_s;
  float max_s;
  float slider_result;
  color fill_slider;
  int layer;
  int num;
  String tag;
  int row_tag = -1;
```

Figura 245: Atributos principales de la clase HMI_SLIDER

3.19.11. Led

Este elemento sirve como un indicador visual para el sistema SCADA puesto que cuando se encuentra inactivo luce apagado y cuando se activa el elemento cambia a un color más claro simulando el encendido de un Led. Se debe asociar a un Tag de tipo digital de salida. El elemento implementado en AuroraLD Studio se muestra en la Figura 246.

Figura 246: Elemento LED

La implementación de este elemento se realiza en base a la clase HMI_LED cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.

- **y:** Coordenada Y en pixeles del elemento.
- **col_led:** Color RGB del element.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **led_size:** Tamaño del elemento en pixeles.
- **tag:** Variable de tipo string que almacena el tag asociado.
- **row_tag:** Número de fila del tag en la tabla de datos del programa.

```
public class HMI_LED {

  final PApplet p;
  int x;
  int y;
  color col_led;
  int layer;
  int num;
  float led_size;
  String tag;
  int row_tag = -1;
```

Figura 247: Atributos principales de la clase HMI_LED

3.19.12. Arco (Arc)

Este elemento permite dibujar una circunferencia con un ángulo inicial y un ángulo final. Tanto los puntos finales e iniciales se pueden unir de dos formas: con una línea recta (cerrando la figura) o creando un arco con un punto central común equidistante al punto final e inicial. En la Figura 248 se ve el elemento arco con los puntos finales e iniciales unidos a un punto central equidistante.

Figura 248: Elemento Arco

El elemento se implementa en base a una clase denominada HMI_ARC cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_arc:** Color RGB del elemento.
- **arc_stroke:** Color RGB del contorno del elemento.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **r:** Radio del elemento.
- **arc_ang1:** Ángulo inicial.
- **arc_ang2:** Ángulo final.

```
public class HMI_ARC {

  final PApplet p;
  int x;
  int y;
  color col_arc;
  color arc_stroke;
  int layer;
  int num;
  float r;
  float arc_ang1;
  float arc_ang2;
```

Figura 249: Atributos principales de la clase HMI_ARC

3.19.13. Turbina eólica (Wind turbine)

Este elemento permite al usuario incluir una turbina eólica en el sistema SCADA como la mostrada en la Figura 250. A este elemento se asocia un tag de tipo digital, el cual permite activar la turbina haciéndola girar.

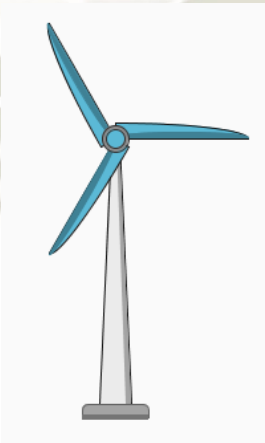


Figura 250: Elemento Turbina eólica

El elemento se implementa en base a la clase HMI_WIND cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_blade:** Color RGB de las aspas.
- **col_tower:** Color RGB de la torre.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **com_wind:** Permite seleccionar entre tener la turbina eólica completa, es decir, torre y aspas o incompleta mostrando solo las aspas.
- **tag:** String que almacena el tag asociado al elemento.
- **row_tag:** Número de final en el que se encuentra el tag en la tabla de datos del programa.

```
public class HMI_WIND {

  final PApplet p;
  int x;
  int y;
  color col_blade;
  color col_tower;
  int layer;
  int num;
  boolean com_wind = true;
  String tag;
  int row_tag = -1;
```

Figura 251: Atributos principales de la clase HMI_WIND

3.19.14. Ventilador (Fan)

Este elemento permite al usuario agregar un ventilador al sistema SCADA. Este ventilador gira y se muestra en funcionamiento si el tag asociado se activa. Los tags que se pueden asociar a este elemento deben ser de tipo digital puesto que solo representa el encendido o apagado.

Figura 252: Elemento Ventilador

La implementación de este elemento se basa en la clase HMI_FAN cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_blade:** Color RGB de las aspas del ventilador.
- **col_bg:** Color RGB del fondo del ventilador.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **com_fan:** Permite elegir entre tener toda la carcasa del ventilador o solamente el ventilador.
- **tag:** String que almacena el tag asociado al elemento.
- **row_tag:** Número de fila en la que se ubica el tag en la tabla de datos del programa.

```
public class HMI_FAN {

  final PApplet p;
  int x;
  int y;
  color col_blade;
  color col_bg;
  int layer;
  int num;
  boolean com_fan = true;
  String tag;
  int row_tag = -1;
```

Figura 253: Atributos principales de la clase HMI_FAN

3.19.15. Torre de enfriamiento (Cooling tower)

La torre de enfriamiento en un elemento que no posee ningún estado de activación y por lo tanto no requiere que se asocie a un tag. Al ser agregado en el sistema SCADA se dibuja una torre de enfriamiento y con un color definido.

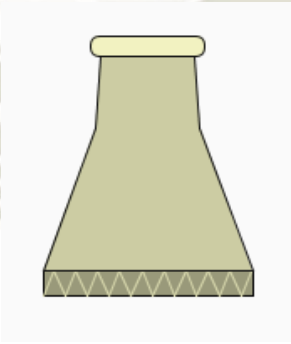


Figura 254: Elemento Torre de enfriamiento

La implementación del elemento torre de enfriamiento se basa en la clase HMI_COOL cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.

- **y:** Coordenada Y en pixeles del elemento.
- **col_bol:** Color RGB del elemento.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **cool_size:** Tamaño del elemento en pixeles.

```
public class HMI_COOL {

  final PApplet p;
  int x;
  int y;
  color col_cool;
  int layer;
  int num;
  float cool_size;
```

Figura 255: Atributos principales de la clase HMI_COOL

3.19.16. PLC

Este elemento simula un PLC pequeño con pocas entradas y salidas. Cuando el sistema SCADA entra en funcionamiento simula la operación de un PLC cambiando el estado de sus entradas y salidas de forma aleatoria.

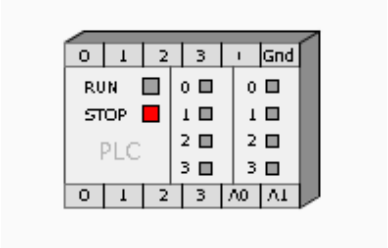


Figura 256: Elemento PLC

Se implementa este elemento en base a la clase HMI_PLC cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_plc:** Color de los Leds indicadores del elemento.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.

```
public class HMI_PLC {

  final PApplet p;
  int x;
  int y;
  color col_plc;
  int layer;
  int num;
```

Figura 257: Atributo principal de la clase HMI_PLC

3.19.17. Tiempo (Time)

Este elemento muestra un reloj basado en el reloj del computador. Solo permite la visualización de horas, minutos y segundos. La función principal de este elemento es solo indicar la hora actual.

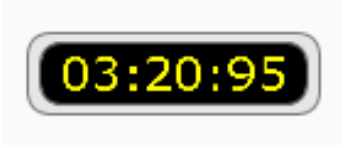


Figura 258: Elemento Tiempo

El elemento tiempo se implementa en base a la clase HMI_TIME cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_text:** Color RGB del texto.
- **col_bg:** Color RGB del fondo.
- **size:** Tamaño de la letra en pixeles.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **wr:** Atributo interno que determina el ancho del rectángulo.
- **hr:** Atributo interno que determina el alto del rectángulo.

```
public class HMI_TIME {

  final PApplet p;
  int x;
  int y;
  color col_text;
  color col_bg;
  float size;
  int layer;
  int num;
  float wr;
  float hr;
```

Figura 259: Atributos principales de la clase HMI_TIME

3.19.18. Botella (Bottle)

Este elemento permite al usuario agregar una botella al sistema SCADA como la mostrada en la Figura 260. A este elemento se asocian tags digitales para colocar la tapa a la botella y tags analógicos que simulen el llenado de la misma.

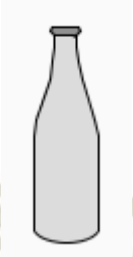


Figura 260: Elemento Botella

La implementación de este elemento se basa en la clase HMI_BOTTLE cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_bottle:** Color RGB del elemento.
- **col_fill:** Color RGB del líquido en la botella.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **min:** Valor mínimo que puede asumir el tag asociado al llenado.
- **max:** Valor máximo que puede asumir el tag asociado al llenado.
- **tag:** String que hace referencia al tag asociado con la animación de llenado por defecto.
- **tag2:** String que hace referencia al tag asociado con la animación de visibilidad para la tapa de la botella.

- **row_tag:** Número de fila en el que se encuentra el tag en la tabla de datos del programa.
- **row_tag2:** Número de final en el que se encuentra el tag2 en la tabla de datos del programa.

```
public class HMI_BOTTLE {

  final PApplet p;
  int x;
  int y;
  color col_bottle;
  color col_fill;
  int layer;
  int num;
  float min;
  float max;
  String tag;
  String tag2;
  int row_tag = -1;
  int row_tag2 = -1;
```

Figura 261: Atributos principales de la clase HMI_BOTTLE

3.19.19. Motor (Motor)

El elemento motor posee una animación de cambio de color por defecto para indicar el estado del motor. Se asocia con tags del tipo digital solamente.

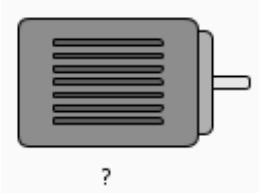


Figura 262: Elemento Motor

La implementación de este elemento se basa en la clase HMI_MOTOR cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_motor_on:** Color RGB del elemento que indica que se encuentra activado.
- **col_motor_off:** Color RGB del elemento que indica que se encuentra desactivado.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **motor_type:** En el caso del elemento motor el tipo hace referencia al ángulo en el que se muestra el elemento.
- **tag:** Variable tipo string que almacena el tag asociado al elemento.
- **row_tag:** Número de fila en la que encuentra el tag en la tabla de datos del programa.

```
public class HMI_MOTOR {

  final PApplet p;
  int x;
  int y;
  color col_motor_on;
  color col_motor_off;
  int layer;
  int num;
  int motor_type = 0; //0
  String tag;
  int row_tag = -1;
```

Figura 263: Atributos principales de la clase HMI_MOTOR

3.19.20. Bomba (Pump)

Este elemento permite agregar una bomba al sistema SCADA. Posee una animación de rotación para los álabes de la bomba y se puede activar esta animación cuando se activa el tag asociado. Solo se pueden asociar tags de tipo digital.

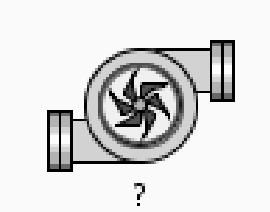


Figura 264: Elemento Bomba

La implementación de este elemento se basa en la clase HMI_PUMP cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_fluid:** Color RGB del fluido.
- **col_blade:** Color RGB de los álabes.
- **col_pump:** Color RGB del elemento.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **dr:** Diámetro de la tubería de salida de la bomba.

```
public class HMI_PUMP {

  final PApplet p;
  int x;
  int y;
  color col_fluid;
  color col_blade;
  color col_pump;
  int layer;
  int num;
  float dr = 25;
```

Figura 265: Atributos principales de la clase HMI_PUMP

3.19.21. Tanques (Tanks)

El elemento tanque permite simular la presencia de líquido en un tanque mediante una animación definida por defecto en el mismo elemento. Para hacer uso de esta animación de llenado es necesario asociar al elemento un tag de tipo analógico. Asimismo, este elemento cuenta con 3 tipos diferentes de tanques, los cuales se pueden apreciar en la Figura 266.

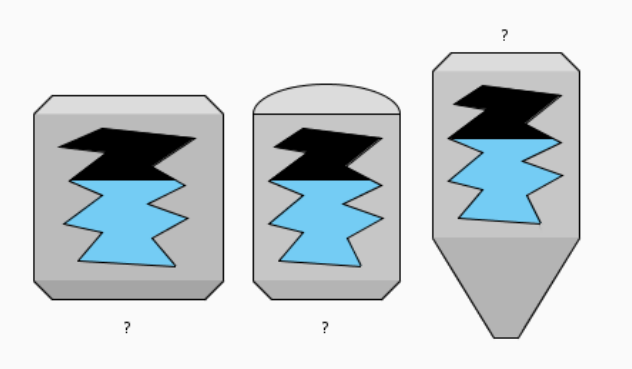


Figura 266: Elemento Tanques

La implementación de este elemento se basa en la clase HMI_TANK cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_tank:** Color RGB del tanque.
- **col_fluid:** Color RGB del fluido.
- **layer:** Número de capa en el que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **tank_type:** Existen 3 tipos de tanques que se puede escoger tal como se muestra en la Figura 266.
- **mint:** Mínimo valor que puede tomar el tag asociado.
- **maxt:** Máximo valor que puede tomar el tag asociado.
- **tag:** Variable tipo string que almacena el tag asociado al elemento.
- **row_tag:** Número de fila en la que se encuentra el tag en la tabla de datos del programa.

```
public class HMI_TANK {

  final PApplet p;
  int x;
  int y;
  color col_tank;
  color col_fluid;
  int layer;
  int num;
  int tank_type = 0;
  float mint = 0;
  float maxt = 0;
  String tag;
  int row_tag = -1;
```

Figura 267: Atributos principales de la clase HMI_TANK

3.19.22. Tuberías (Pipes)

Este elemento permite al usuario agregar tuberías al sistema SCADA. Se puede modificar el color, la longitud, el tipo, el ángulo, etc.

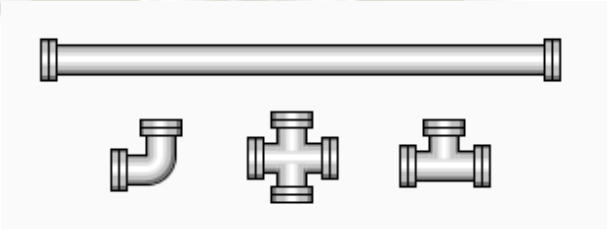


Figura 268: Elemento Tubería

La implementación de este elemento se realiza en base a la clase HMI_PIPE cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.

- **col_pipe:** Color RGB de la tubería.
- **col_fluid:** Color RGB del fluido.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **lr:** Longitud de la tubería en pixeles.
- **dr:** Diámetro de la tubería en pixeles.
- **pipe_type:** Los tipos de tubería se muestran en la Figura 268. Se pueden escoger 4 tipos diferentes de tuberías y/o uniones.
- **pipe_angle:** Permite seleccionar la orientación que tendrá el elemento escogiendo valores de 0°, 90°, 180° y 270°.
- **tag:** Variable tipo string que almacena el tag asociado al elemento.
- **row_tag:** Número de fila donde se encuentra el tag en la tabla de datos del programa.

```
public class HMI_PIPE {

  final PApplet p;
  int x;
  int y;
  color col_pipe;
  color col_fluid;
  int layer;
  int num;
  float lr;
  float dr;
  int pipe_type = 0;
  int pipe_angle = 0;
  String tag;
  int row_tag = -1;
```

Figura 269: Atributos principales de la clase HMI_PIPE

3.19.23. Válvulas (Valves)

Este elemento permite dibujar válvulas como las mostradas en la Figura 270. Estas válvulas se emplean por lo general junto con los elementos tanque y/o tuberías.

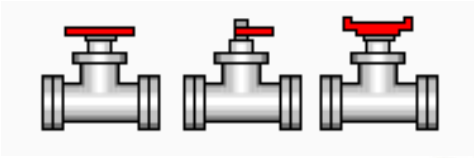


Figura 270: Elemento Válvula

Se implementa este elemento en base a la clase denominada HMI_VALVE cuyos atributos principales se muestran a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_pipe:** Color RGB de la tubería.
- **col_fluid:** Color RGB del fluido.
- **col_valve:** Color RGB de la válvula.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **dr:** Hace referencia al diámetro de la tubería en pixeles.
- **valve_type:** El tipo de válvula hace referencia a la forma que tendrá esta. Se puede apreciar en la Figura 270 los 3 tipos de válvulas que se pueden escoger.
- **valve_angle:** El ángulo de la válvula puede ser 0°, 90°, 180° y 270°.
- **tag:** Variable de tipo string que almacena el tag asociado al elemento.
- **row_tag:** Número de fila en la que se encuentra el tag en la tabla de datos del programa.

```
public class HMI_VALVE {

  final PApplet p;
  int x;
  int y;
  color col_pipe;
  color col_fluid;
  color col_valve;
  int layer;
  int num;
  float dr;
  int valve_type = 0;
  int valve_angle = 0;
  String tag;
  int row_tag = -1;
```

Figura 271: Atributos principales de la clase HMI_VALVE

3.19.24. Edificios (Buildings)

Este elemento permite al usuario agregar edificios al entorno SCADA que pueden ser activados solo por variables boolenas para encender las luces. Entre los parámetros configurables se tiene el tag asociado a las luces, color de las luces, color del elemento, etc.

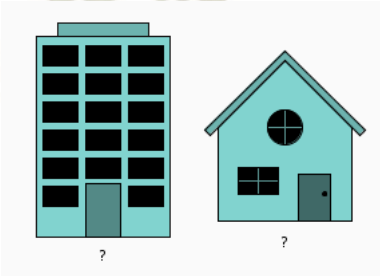


Figura 272: Elemento Edificio

Se implementa este elemento en base a la clase denominada HMI_BUILD cuyos atributos principales se muestran a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_build:** Color del elemento.
- **col_light:** Color de la luz de los edificios.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **build_type:** Tipo de elemento, es decir, tipo edificio o casa.
- **tag:** Variable tipo string que almacena el tag asociado al elemento.
- **row_tag:** Número de fila en la que se ubica el tag dentro de la tabla de datos del programa.

```
public class HMI_BUILD {

  final PApplet p;
  int x;
  int y;
  color col_build;
  color col_light;
  int layer;
  int num;
  boolean build_type = false;
  String tag;
  int row_tag = -1;
```

Figura 273: Atributos principales de la clase HMI_BUILD

3.19.25. Torre de transmisión (Transmission tower)

El elemento torre de transmisión permite dibujar una torre de transmisión como la mostrada en la Figura 274. El único parámetro alterable de este elemento es el color.

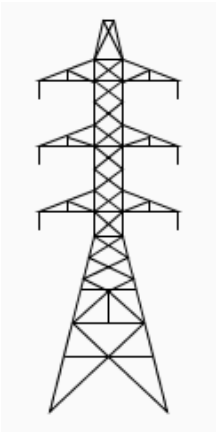


Figura 274: Elemento Torre de transmisión

Se implementa este elemento en base a una clase denominada HMI_POWER cuyos atributos principales se detallan a continuación:

- **x:** Hace referencia a la coordenada X del elemento en pixeles.
- **y:** Hace referencia a la coordenada Y del elemento en pixeles.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **col_power:** Color RGB del elemento.

```
public class HMI_POWER {

  final PApplet p;
  int x;
  int y;
  int layer;
  int num;
  color col_power;
```

Figura 275: Atributos principales de la clase HMI_POWER

3.19.26. Señales (Signs)

En AuroraLD Studio el elemento señales permiten al usuario agregar señales predefinidas tales como peligro, riesgo eléctrico, radiación, stop, etc. Estas señales están construidas en base a imágenes png ubicadas en la carpeta data. Al arrastrar este elemento en el espacio de trabajo SCADA se muestra una señal por defecto, pero puede ser cambiada desde los parámetros del elemento.

Figura 276: Elemento Señal

Se implementa este elemento en base a la clase denominada HMI_SIGNS cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X del elemento en pixeles.
- **y:** Coordenada Y del elemento en pixeles.
- **layer:** Capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **size:** Tamaño del elemento.
- **sign_type:** Tipo de señal a mostrar. Se han definido las imágenes previamente en el código y se ha a signado un valor entero cada una. Esta variable representa esos valores enteros y por ende permite la selección de las señales.

```
public class HMI_SIGNS {

  final PApplet p;
  int x;
  int y;
  int layer;
  int num;
  float size;
  int sign_type = 0;
```

Figura 277: Atributos principales de la clase HMI_SIGNS

3.19.27. Medidor analógico (Analog meter)

El elemento medidor analógico permite mostrar variables analógicas tanto en forma circular como en forma de barra. Se puede modificar los parámetros para mostrar la información en el rango deseado.

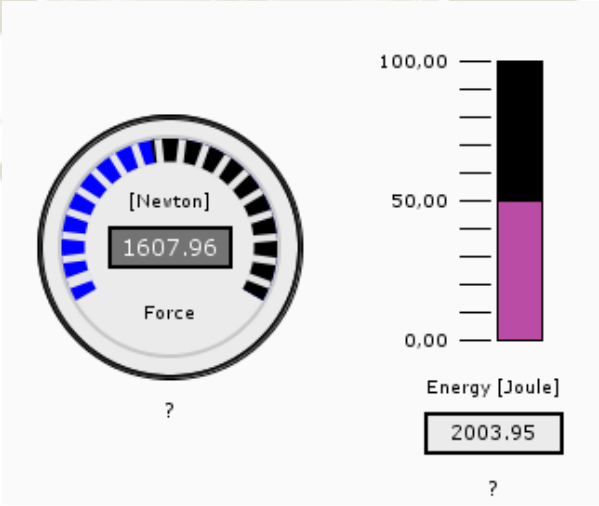


Figura 278: Elemento Medidor analógico

Este elemento se implementa en base a una clase denominada HMI_METER cuyos atributos principales se detallan a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **col_value:** Color RGB de medición en el instrumento analógico.
- **layer:** Número de capa en el cual se dibuja el elemento.
- **num:** Identificador único del elemento.
- **meter_type:** Tipo de medidor analógico: barra o circular.
- **value:**
- **meter_value:**
- **minm:** Mínimo rango que puede asumir el tag asociado al elemento.
- **maxm:** Máximo rango que puede asumir el tag asociado al elemento.
- **unit:** Variable tipo string que hace referencia a la unidad.
- **parameters:** Variable tipo string que hace referencia al nombre de la unidad.
- **tag:** Variable tipo string que hace referencia al tag asociado al elemento.
- **row_tag;** Número de final en la que se encuentra el tag en la tabla de datos del programa.

```
public class HMI_METER {

  final PApplet p;
  int x;
  int y;
  color col_value;
  int layer;
  int num;
  int meter_type;
  float value;
  float meter_value;
  float minm;
  float maxm;
  String unit;
  String parameter;
```

Figura 279: Atributos principales de la clase HMI_METER

3.19.28. Alarmas (Alarms)

El elemento alarmas alerta al usuario de la ocurrencia de algún evento mediante 5 distintos sonidos según se haya configurado. Las alarmas en AuroraLD Studio funcionan tanto para variables analógicas como para variables digitales. El sistema SCADA en modo edición muestra la imagen de la alarma, pero cuando el sistema se ejecuta esta desaparece y solamente vuelve a aparecer si la alarma se activa generando así una alerta sonora y gráfica.

Figura 280: Elemento Alarma

Este elemento se implementa en base a una clase denominada HMI_ALARM cuyos atributos principales se muestran a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **layer:** Número de capa en la que se realiza el dibujo del elemento.
- **num:** Identificador único del elemento.
- **row_tag:** Número de fila del tag en la tabla de datos del programa.
- **alarm_sound:** Variable entera que hace referencia al sonido que tendrá la alarma. Se puede escoger 5 distintos sonidos.
- **alarm_action:** Hace referencia a la acción que debe cumplirse para activar la alarma. Se tiene 4 tipos distintos: activa cuando el tag pasa a verdadero, activa cuando el tag pasa a falso, activa si el tag está activo o activa si el tag está inactivo.
- **tag:** Variable tipo string que hace referencia al tag asociado al elemento.
- **alarm_type:** El tipo de alarma puede ser digital o analógico en base al tag.
- **n1:** Primer valor del intervalo (variables analógicas).
- **n2:** Segundo valor del intervalo (variables analógicas).
- **n3:** Tercer valor del intervalo (variables analógicas).
- **n4:** Cuarto valor del intervalo (variables analógicas).
- **s1:** Símbolo del primer intervalo.
- **s2:** Símbolo del segundo intervalo.
- **s3:** Símbolo del tercer intervalo.
- **s4:** Símbolo del cuarto intervalo.
- **num_interval:** Número de intervalos.
- **msg1:** Mensaje del primer intervalo.

- **msg2:** Mensaje del segundo intervalo.
- **msg3:** Mensaje del tercer intervalo.
- **msg4:** Mensaje del cuarto intervalo.

```
public class HMI_ALARM {

  final PApplet p;
  int x;
  int y;
  int layer;
  int num;
  int row_tag = -1;
  int alarm_sound = 0; //0 ALA
  int alarm_action = 0; //0 CH
  String tag;
  boolean alarm_type = false;
  float n1 = 0; //FIRST NUMBER
  float n2 = 0; //SECOND NUMBE
  float n3 = 0; //THIRD NUMBER
  float n4 = 0; //FOURTH NUMBE
  int s1 = 0; //0 >= - 1 <=
  int s2 = 0; //0 >= - 1 <=
  int s3 = 0; //0 >= - 1 <=
  int s4 = 0; //0 >= - 1 <=
  int num_interval = 1;
  String msg1;
  String msg2;
  String msg3;
  String msg4;
```

Figura 281: Atributos principales de la clase HMI_ALARM

3.19.29. Gráfico de tendencias (Trends)

Este elemento permite al usuario monitorear como máximo 3 variables analógicas o digitales en un mismo gráfico. Se selecciona un color distinto y un tag asociado a estos colores para configurar el elemento. Asimismo, también se incluye una opción para escoger la ventana que se visualizará, es decir, el tiempo máximo que se verá en todo momento. Los rangos permitidos de la ventana van desde 1 hasta 60 segundos.

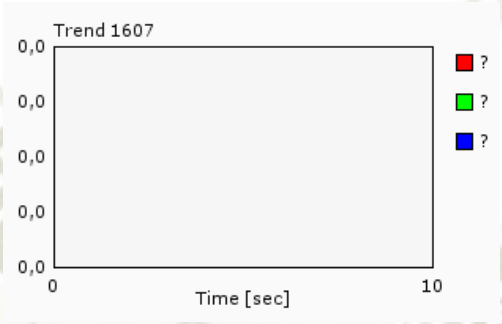


Figura 282: Elemento Trends

Este elemento se implementa en base a una clase denominada HMI_GRAPH cuyos atributos principales se muestran a continuación:

- **x:** Coordenada X en pixeles del elemento.
- **y:** Coordenada Y en pixeles del elemento.
- **layer:** Número de capa en la que se dibuja el elemento.
- **num:** Identificador único del elemento.
- **wr:** Ancho del elemento en pixeles.
- **hr:** Alto del elemento en pixeles.
- **col_bg:** Color RGB del fondo.
- **col_l1:** Color RGB de la primera variable a graficar.
- **col_l2:** Color RGB de la segunda variable a graficar.
- **col_l3:** Color RGB de la tercera variable a graficar.

- **tag1:** Tag de la primera variable.
- **tag2:** Tag de la segunda variable.
- **tag3:** Tag de la tercera variable.
- **row_tag1:** Fila en la que se encuentra el tag de la primera variable en la tabla de datos del programa.
- **row_tag2:** Fila en la que se encuentra el tag de la segunda variable en la tabla de datos del programa.
- **row_tag3:** Fila en la que se encuentra el tag de la tercera variable en la tabla de datos del programa.
- **window_size:** Tamaño de la ventana en segundos. Rango de 1 segundo hasta 60 segundos.
- **trend_title:** Título de la gráfica.

```
public class HMI_GRAPH {

  final PApplet p;
  int x;
  int y;
  int layer;
  int num;
  float wr;
  float hr;
  color col_bg;
  color col_l1;
  color col_l2;
  color col_l3;
  String tag1;
  String tag2;
  String tag3;
  int row_tag1 = -1;
  int row_tag2 = -1;
  int row_tag3 = -1;
  int window_size;
  String trend_title;
```

Figura 283: Atributos principales de la clase HMI_GRAPH

3.19.30. Imágenes (Image)

El elemento Image permite al usuario cargar una imagen que esté en formato JPG, PNG o GIF en el entorno de trabajo SCADA con la finalidad de agregar algún elemento que no se encuentre predefinido o incluso mostrar la planta real. Al agregar este elemento en el entorno SCADA se muestra una casilla en blanco, pero una vez que se selecciona la imagen que se desea cargar se mostrará automáticamente y será agregada junto con los archivos del proyecto.

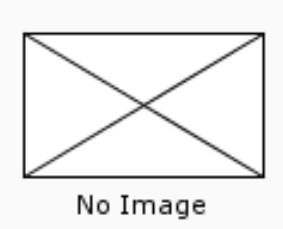


Figura 284: Elemento Imagen

La carga de la imagen se realiza mediante un JFileChooser tal como se muestra en la Figura 285.

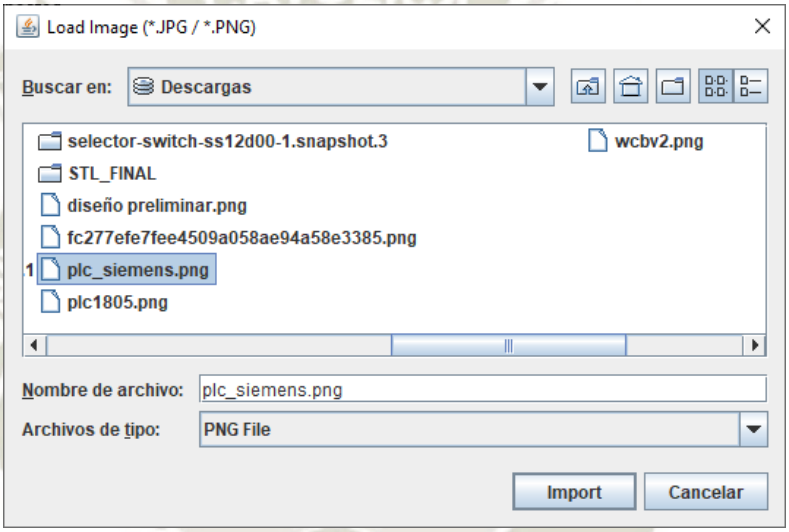


Figura 285: Carga de una imagen en AuroraLD Studio

Una vez que se ha seleccionado importar la imagen aparecerá en el entorno SCADA. En la Figura 286 se muestra la imagen de un PLC siemens que ha sido agregada a AuroraLD Studio.

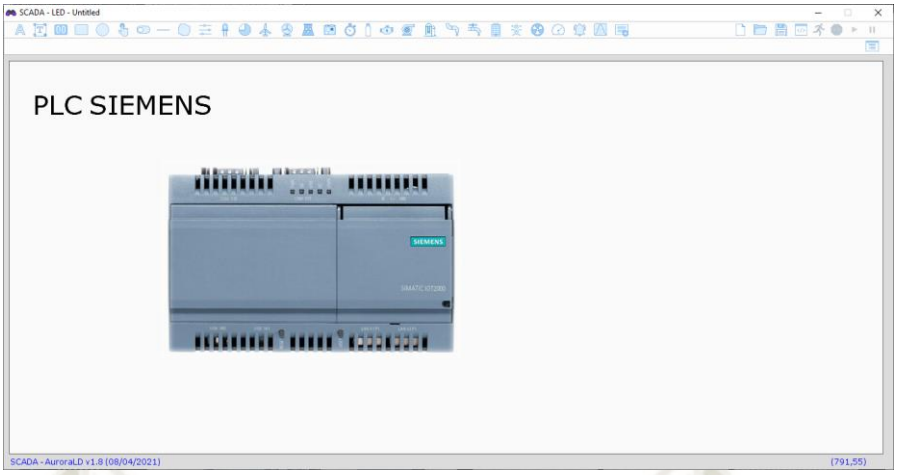


Figura 286: Ejemplo de imagen cargada en el entorno SCADA

Se implementa este elemento en base a la clase HMI_IMAGE cuyos atributos se muestran a continuación:

```
public class HMI_IMAGE {

  final PApplet p;
  int x;
  int y;
  int layer;
  int num;
  float wr = 100;
  float hr = 60;
  String image_file;
  float image_percentage;
  PImage picture;
```

Figura 287: Atributos principales de la clase HIMI_IMAGE

- **x:** Representa la coordenada X en pixeles del elemento.
- **y:** Representa la coordenada Y en pixeles del elemento.
- **layer:** Indica en que capa se debe dibujar el elemento. De acuerdo a este valor los elementos podrían estar detrás o delante de otros elementos.

- **num:** Identificador único de cada elemento. Su valor incrementa conforme se agregan más elementos del mismo tipo.
- **image_file:** Variable de tipo PImage la cual almacena la imagen que se escoja para cargar en el entorno SCADA.
- **image_percentage:** Indica el tamaño de la imagen en porcentaje, siendo el 100% el tamaño original de la imagen.

Por otro lado, el entorno SCADA también cuenta con herramientas tales como: Nuevo, Abrir, Guardar, Compilar, Ejecutar, Detener, Play, Pausa y Log de Alarmas. Estas herramientas se pueden observar en la parte superior derecha del entorno.

Figura 288 - Herramientas entorno SCADA

También encontramos en la parte superior derecha el log de alarmas que lleva el registro de todas las alarmas que se han activado durante el proceso. El icono para poder acceder al log de alarmas de muestra en la Figura 289.

Figura 289 - Log de Alarmas

Finalmente, en la parte inferior del entorno SCADA se observa la versión del software en el lado izquierdo y en el lado derecho se puede observar las coordenadas “X” y “Y” del cursor en el entorno de trabajo SCADA (Figura 290). Estas coordenadas sirven como guías para colocar los distintos elementos.

Figura 290 - Barra inferior entorno SCADA

3.20. Alarmas SCADA

Cuando se realiza la supervisión de un proceso algunas veces se requiere tener un indicador que se activa cuando se cumplen ciertas condiciones. En los diferentes Sistemas de Supervisión y Adquisición de Datos (SCADA) se colocan alarmas asociados a los Tags del proceso. La función básica de una alarma es avisar al supervisor de un evento ocurrido en el proceso mediante un sonido y un evento (previamente configurado).

Los eventos se generan junto con los elementos Alarm en AuroraLD Studio y se almacenan en una tabla denominada Alarm Log pudiendo así almacenar hasta 100 eventos diferentes antes de que se reinicie la tabla.

El Alarm Log muestra información previamente definida que siempre acompaña a un determinado evento, entre esta información se tiene:

- **Número de evento:** Se asigna un valor numérico de forma automática dependiendo de cuantos eventos se van registrando en el registro de alarmas.
- **Tiempo:** Hace referencia a la hora en la que ocurrió un determinado evento.
- **Fecha:** Hace referencia a la fecha en la que ocurre un determinado evento.
- **Tag:** Hace referencia al tag asociado con dicha alarma y evento.
- **Evento:** Breve resumen del evento.
- **Descripción:** Descripción breve acerca del evento.

El alarm log o registro de alarmas implementado en AuroraLD Studio se muestra en la Figura 291 y se conforma por columnas con la información previamente detallada.

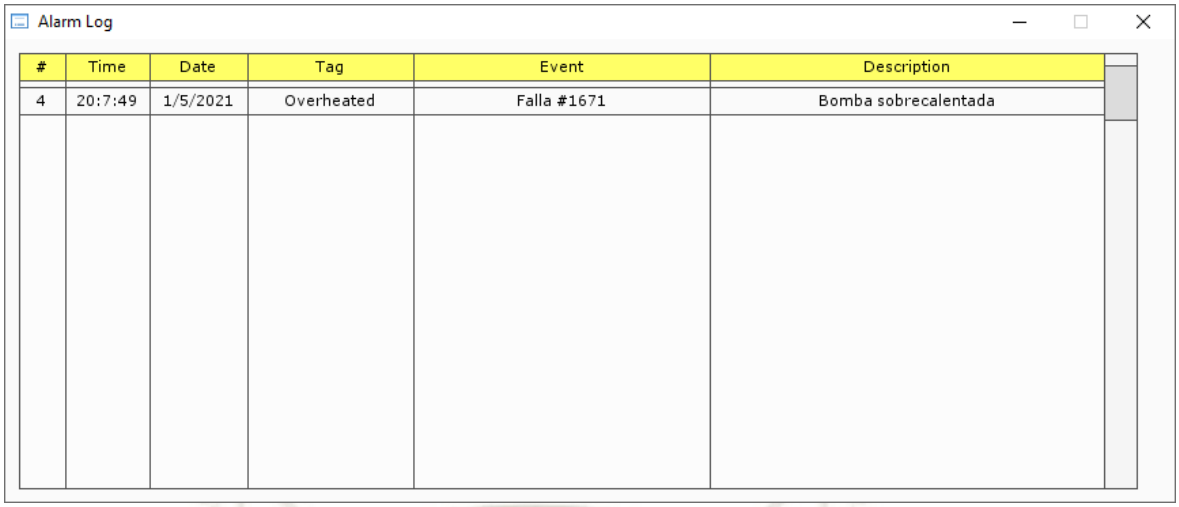


Figura 291: Log de alarmas

Como se detalló antes el elemento Alarma (Alarm) se caracteriza por asociarse a un tag y producir un sonido cuando se cumplen las condiciones especificadas en sus parámetros. Existen 4 modos de configuración para las Alarmas y se puede trabajar tanto en modo digital como analógico.

Entre los modos que se pueden especificar para las alarmas se tiene:

- **Change to on (Cambio a encendido):** La alarma se dispara con un flanco de subida de acuerdo al tag que se tenga asociado.
- **Change to off (Cambio a apagado):** La alarma se dispara con un flanco de bajada de acuerdo al tag que se tenga asociado.
- **On (Encendido):** La alarma se activa siempre y cuando el tag asociado se encuentre encendido.
- **Off (Apagado):** La alarma se activa siempre y cuando el tag asociado se encuentre apagado.

Los modos de trabajo especificados se pueden cambiar accediendo a la ventana de parámetros del elemento alarma. En la Figura 292 se tiene una alarma de tipo digital asociada a un tag “Overheated”, se ha especificado también el tipo de sonido a reproducir y el modo de trabajo de la alarma ha sido configurado para activar se con un flanco de subida.

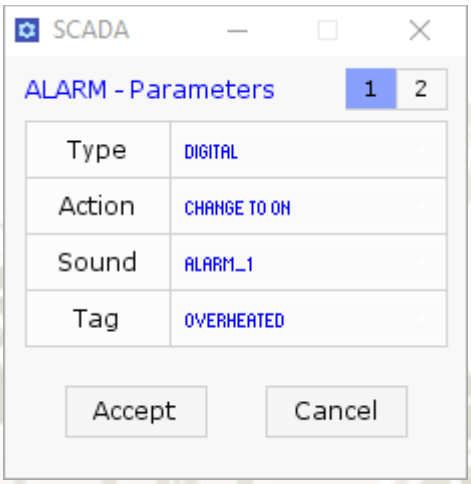


Figura 292: Parámetros de Alarmas

La configuración adecuada de las alarmas crea un entorno más fácil de monitorear conociendo así el cambio de determinadas variables en el momento que se producen dichos cambios y tomar decisiones en caso se necesite.

La implementación del registro de alarmas y eventos se logra a través de la creación de una ventana adicional y declarando variables para cada una de las columnas mostradas en la ventana. Las variables empleadas para cada una de las columnas se muestran en el código de la Figura 293.

```
String[] al_time = new String[100];
String[] al_date = new String[100];
String[] al_tag = new String[100];
String[] al_event = new String[100];
String[] al_description = new String[100];

WINDOW_LOG LOG;

class WINDOW_LOG extends PApplet {

  boolean slide = false;
  int ppy = 0;
  float speed = 0;

  WINDOW_LOG() {
    super();
    PApplet.runSketch(new String[] {this.getClass().getSimpleName()}, this);
    surface.setTitle("Alarm Log");
  }
```

Figura 293: Parámetros principales del registro de alarmas y eventos

3.21. Parámetros SCADA

Se accede a los parámetros de los elementos SCADA haciendo click derecho en los elementos y seleccionando la opción parámetros tal como se muestra en la Figura 294.

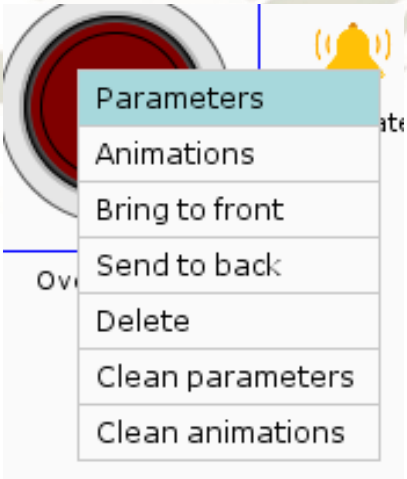


Figura 294: Opción click derecho en elementos

La opción parámetros abrirá una ventana distinta dependiendo del tipo de elemento, pero el funcionamiento es el mismo en todos los casos. Los parámetros que se pueden modificar son en su mayoría atributos principales de cada elemento. Se muestra un ejemplo de la ventana de parámetros para el elemento LED en la Figura 295.

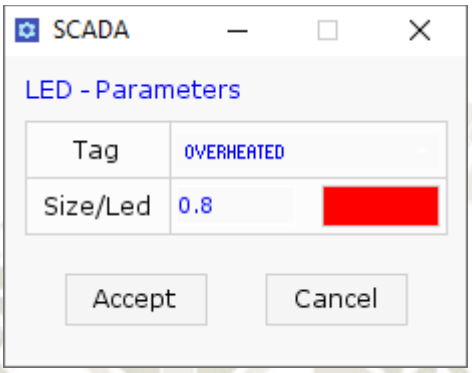


Figura 295: Ejemplo de ventana de parámetros LED

El funcionamiento de la ventana de parámetros se detalla mejor explicando el algoritmo implementado:

- Se muestra una ventana con los parámetros principales de un elemento determinado.
- Se seleccionan y colocan los parámetros que se deseen para un elemento determinado.
- Al hacer click en la opción aceptar estos parámetros se asignan directamente en los atributos de cada clase tomando en cuenta el identificador del elemento (valor único) y así cambian los parámetros de un determinado elemento.

3.22. Animaciones SCADA

Las animaciones en AuroraLD Studio pueden funcionar con variables digitales y analógicas dependiendo de los elementos en donde se configuran las animaciones. Estas animaciones están implementadas con el propósito de hacer más dinámicos los elementos en el entorno SCADA haciendo que sean más vistosos y entendibles por los usuarios.

Se accede a este menú de animaciones al hacer click derecho en un elemento y posteriormente escoger la opción Animaciones, tal como se muestra en la Figura 296.

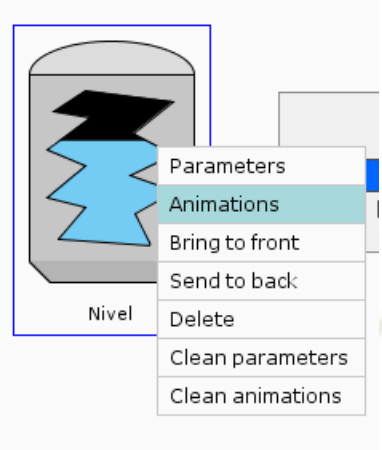


Figura 296: Opciones Click Derecho Elementos

Esta opción abre la ventana que permite configurar las animaciones de un determinado elemento:

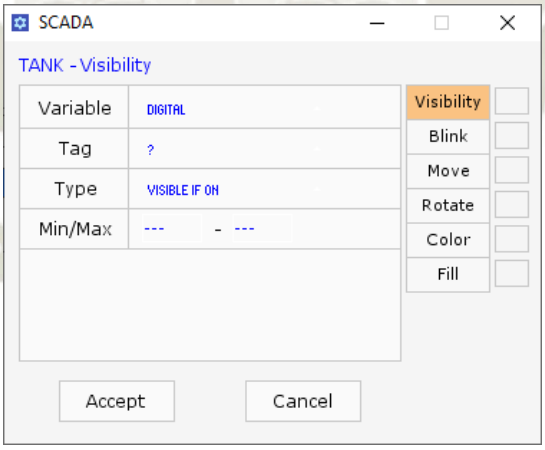


Figura 297: Configuración de animaciones

Notamos en la Figura 297 que en la ventana de configuración de animaciones se tiene al derecho todas las animaciones posibles a ser configuradas en un elemento SCADA. Las animaciones con las que cuenta AuroraLD Studio para el entorno SCADA son:

- Visibilidad (Visibility)
- Parpadeo (Blink)
- Mover (Move)
- Rotar (Rotate)
- Color
- Llenado (Fill)

No todas las animaciones mencionadas están disponibles en todos los elementos ya que se consideran repetitivos o innecesarios en algunos casos.

3.22.1. Visibilidad (Visibility)

Por defecto todos los elementos que se agregan al entorno SCADA son visibles. Si se requiere que un determinado elemento desaparezca o aparezca cuando se cumplen ciertas condiciones ya sean digitales o analógicas se emplea la animación visibilidad. En la Figura 298 se muestran los parámetros que se pueden configurar para la animación visibilidad.

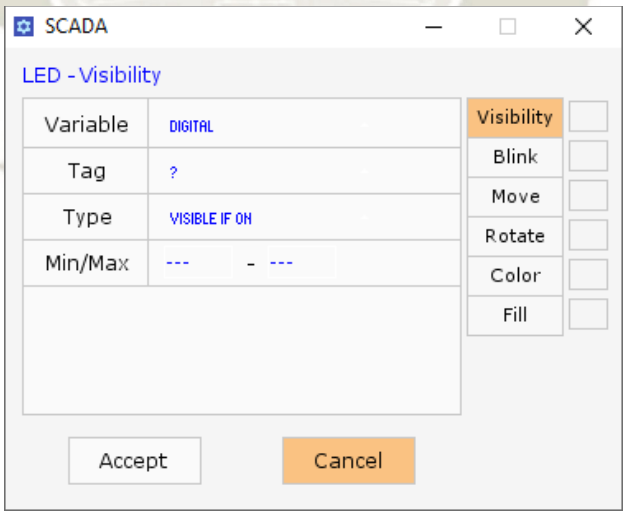


Figura 298: Animación visibilidad

El código que ejecuta la animación se encuentra en el interior de la clase de cada uno de los elementos SCADA y comprende las siguientes variables principales:

- **row_vis:** Hace referencia a la fila en la que se encuentra el tag asociado a la animación de visibilidad en la tabla de datos del programa.
- **tag_vis:** Hace referencia al tag asociado a la animación visibilidad.
- **min_vis:** En el caso de usar la animación para variables del tipo analógico se escoge un valor mínimo desde el cual se empezará a mostrar un determinado elemento.
- **max_vis:** En el caso de usar la animación para variables del tipo analógico se escoge un valor máximo definiendo así el máximo valor que se puede alcanzar para seguir mostrando la imagen.
- **type_vis:** AuroraLD Studio permite escoger dos tipos de animación de visibilidad. Visible si el tag se activa o visible si el tag se encuentra inactivo.
- **var_vis:** Por último, la variable hace referencia a si se trata de un Tag digital o analógico.

Se muestra en la Figura 299 las variables principales que comparten los elementos SCADA. A diferencia de lo detallado en la figura se aprecia el tipo de variable correspondiente a cada una de las variables mencionadas anteriormente.

```
//VISIBILITY
int row_vis = -1;
String tag_vis;
float min_vis;
float max_vis;
String type_vis; //Visible
String var_vis; //Digital or
boolean vis_on = false;
boolean vis_active = false;
```

Figura 299: Variables de la animación visibilidad

Básicamente el algoritmo solo impide el dibujo del elemento mediante una función condicional, es decir, una instrucción IF. Si se cumplen las condiciones configuradas en la animación visibilidad el elemento se mostrará. Esta animación de visibilidad funciona con todos los elementos SCADA.

3.22.2. Parpadeo (Blink)

Ningún elemento agregado al entorno SCADA produce parpadeo cuando el entorno se ejecuta. Esta animación permite crear el efecto de parpadeo en los elementos en los que se configure. La animación de parpadeo se puede configurar para ser ejecutada tanto si se cumplen condiciones digitales o analógicas. La ventana de parámetros que permite la configuración de esta animación se muestra en la Figura 300.

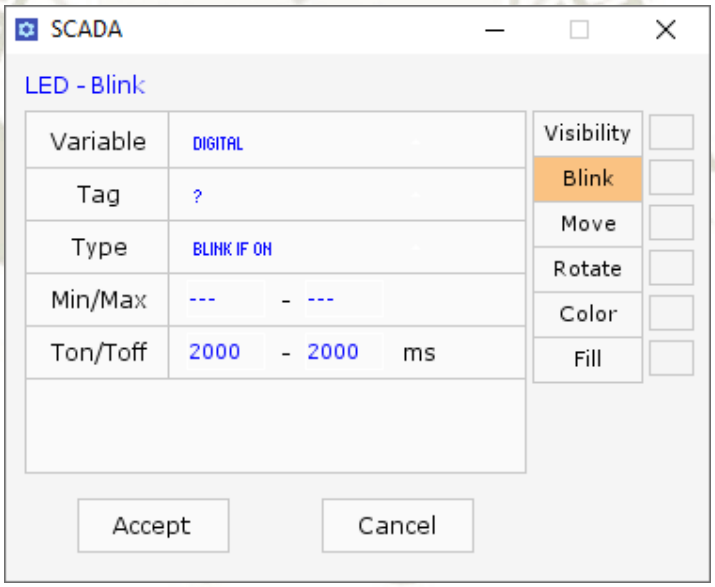


Figura 300: Animación parpadeo

El código que ejecuta la animación se encuentra en el interior de la clase de cada uno de los elementos SCADA y comprende las siguientes variables principales:

- **row_bk:** Indica la fila donde se encuentra el tag dentro de la tabla de datos del programa.

- **tag_bk:** Indica el tag al que está asociado la animación parpadeo.
- **min_bk:** Indica el valor mínimo para activar la animación en el caso de tags analógicos.
- **max_bk:** Indica el valor máximo en el que activa la animación en el caso de tags analógicos. En los tags analógicos se especifica un rango definido por un valor máximo y mínimo para activar la animación.
- **type_bk:** Se puede escoger dos tipos de animación parpadeo los cuales permiten activar la animación cuando el tag se activa o desactiva según se requiera.
- **var_bk:** La animación parpadeo funciona tanto para variables del tipo digital como analógicas.
- **ton_bk:** Se escoge un tiempo en milisegundos para el temporizador denominado tiempo encendido.
- **tof_bk:** Se escoge un tiempo en milisegundos para el temporizador denominado tiempo apagado.
- **atime_bk:** Registra el tiempo actual que viene dado por la función millis().
- **ptime_bk:** Registra el tiempo previo. Con este parámetro y el tiempo actual se puede conocer cuánto tiempo pasa entre un pulso de parpadeo y otro.

```
//BLINK
int row_bk = -1;
String tag_bk;
float min_bk;
float max_bk;
String type_bk; //Blink if
String var_bk; //Digital or
int ton_bk;
int tof_bk;
boolean bk_on = false;
boolean bk_active = false;
int atime_bk;
int ptime_bk;
```

Figura 301: Variables de la animación parpadeo

3.22.3. Mover (Move)

Para agregar movimiento a los elementos del entorno SCADA se emplea la animación Mover. El movimiento de un determinado elemento puede producirse al cumplirse condiciones digitales o analógicas. Es importante mencionar que ningún elemento puede salir fuera del espacio de trabajo en condiciones normales, pero la animación Mover permite salir de dichos rangos mientras el entorno SCADA se encuentre en ejecución. La ventana de parámetros que permite la configuración de la animación mover se muestra en la Figura 302.

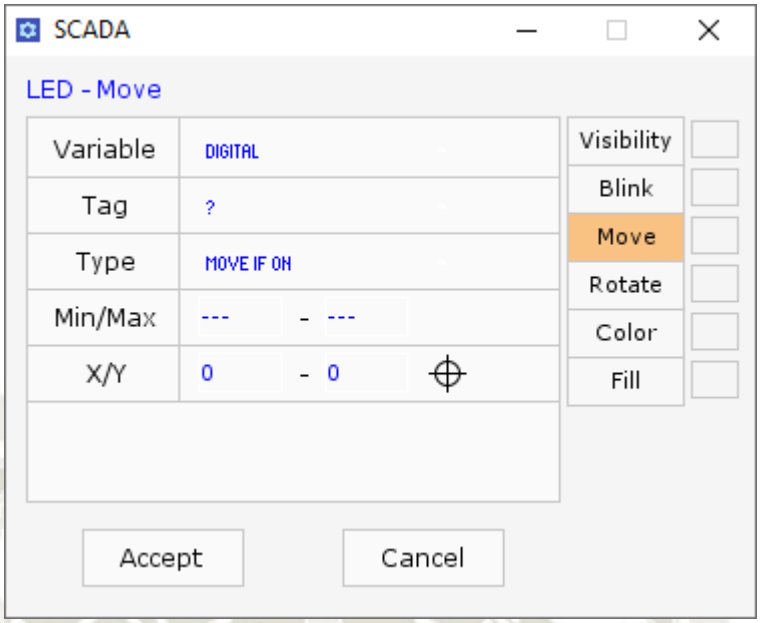


Figura 302: Animación mover

El código que ejecuta la animación se encuentra en el interior de la clase de cada uno de los elementos SCADA y comprende las siguientes variables principales:

- **row_mov:** Hace referencia a la fila en la tabla de datos del programa del tag asociado a la animación
- **tag_mov:** Tag asociado a la animación puede ser de tipo digital o analógico.
- **min_mov:** Si se selecciona un tag analógico se debe especificar un rango de valores para poder linealizar y realizar el movimiento. Esta variable especifica el valor mínimo del rango que puede asumir el tag.
- **max_mov:** Valor máximo que puede asumir el tag.
- **type_mov:** Se pueden escoger dos diferentes tipos de condiciones para activar la animación. Uno permite realizar el movimiento si el tag se encuentra activo y el otro realiza el movimiento si el tag se encuentra inactivo. Esta variable solo aplica si se especifica un tag digital.
- **var_mov:** Tipo de variable, es decir, analógica o digital.

- **xf_mov:** Coordenada X en pixeles del punto final (punto destino).
- **yf_mov:** Coordenada Y en pixeles del punto final (punto destino).
- **tpx_mov:** Desplazamiento en el eje X al linealizar los valores mínimos y máximos del tag cuando se trabaja con variables analógicas.
- **tpy_mov:** Desplazamiento en el eje Y al linealizar los valores mínimos y máximos del tag cuando se trabaja con variables analógicas.

```
//MOVE
int row_mov = -1;
String tag_mov;
float min_mov;
float max_mov;
String type_mov; //Blink if
String var_mov; //Digital or
int xf_mov;
int yf_mov;
boolean mov_on = false;
boolean mov_active = false;
int tpx_mov;
int tpy_mov;
```

Figura 303: Variables de la animación mover

La animación mover incluye una opción que permite previsualizar la posición final y el desplazamiento de un elemento hasta su destino tal como se muestra en la Figura 304. Con esta herramienta es más fácil posicionar los elementos.

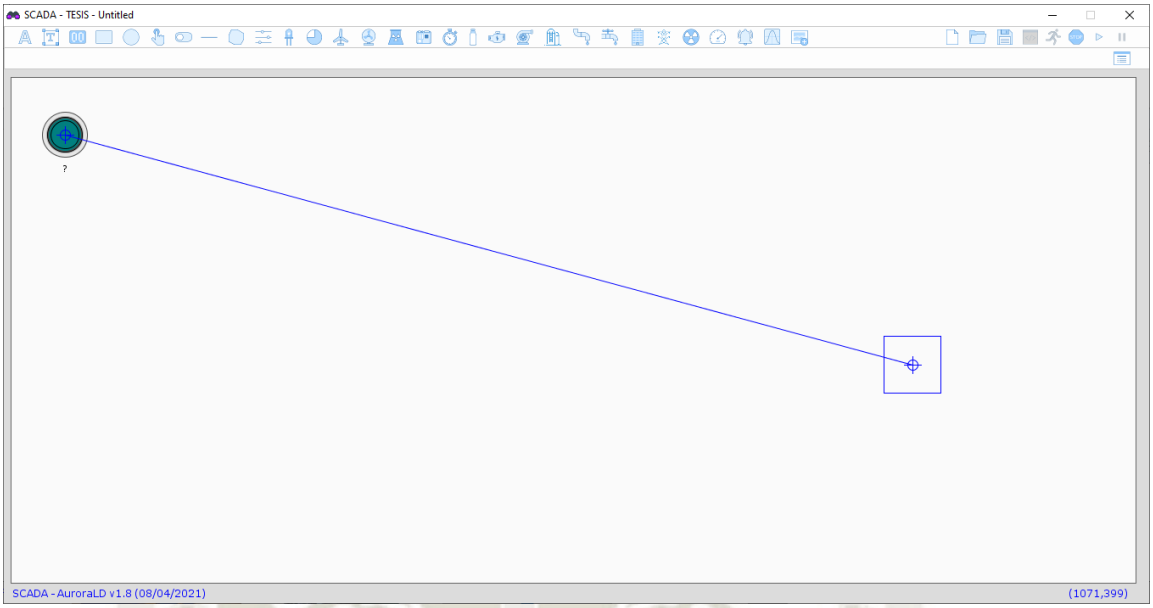


Figura 304: Previsualización de animación mover

3.22.4. Rotar (Rotate)

La animación de rotación brinda la posibilidad de rotar ciertos elementos al cumplirse condiciones digitales o analógicas. Esta animación no se encuentra disponible en todos los elementos ya que en algunos casos es repetitivo o completamente innecesario. La ventana de parámetros que permite la configuración de la animación rotar se muestra en la Figura 305.

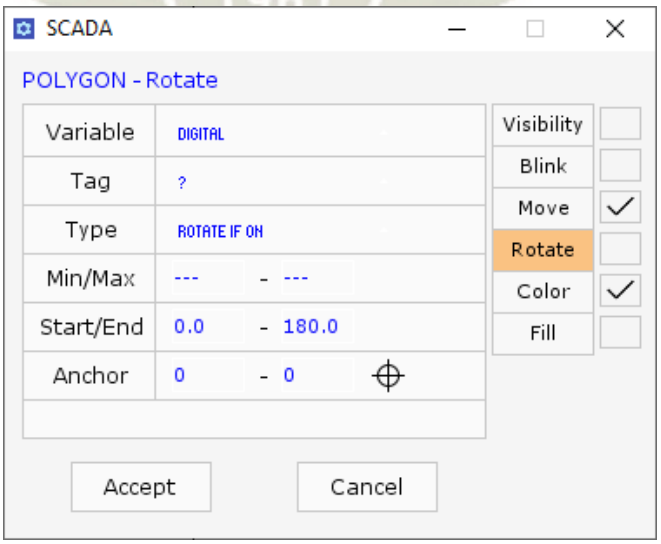


Figura 305: Animación rotar

El código que ejecuta la animación se encuentra en el interior de la clase de los elementos SCADA y comprende las siguientes variables principales:

- **row_rot:** Indica la fila en la tabla de datos del programa del tag asociado a la animación.
- **tag_rot:** Variable de tipo string que hace referencia al tag asociado a la animación.
- **min_rot:** Valor mínimo para activar la animación en caso de variables analógicas.
- **max_rot:** Valor máximo para activar la animación en caso de variables analógicas.
- **type_rot:** En el caso de variables digitales se puede escoger dos tipos de activación para la animación de rotación. Rotar si el tag está activa o rotar si el tag está inactivo.
- **var_rot:** Puede ser digital o analógico.
- **anchorx_rot:** Indica la coordenada X del anchor point.
- **anchory_rot:** Indica la coordenada Y del anchor point.
- **angle_rot:** Indica el ángulo de rotación en función de los valores mínimos y máximos especificados en el caso de variables analógicas.

```
//ROTATE
int row_rot = -1;
String tag_rot;
float min_rot;
float max_rot;
String type_rot;
String var_rot;
float a1_rot;
float a2_rot;
int anchorx_rot;
int anchory_rot;
boolean rot_on = false;
boolean rot_active = false;
float angle_rot = 0;
```

Figura 306: Variables de la animación rotar

La animación rotar cuenta con una herramienta que ayuda a previsualizar el camino de rotación que seguirá el elemento en función de un punto denominado anchor point tal como se muestra en la Figura 307. Todos los elementos tienen una posición XY denominado punto de origen. Si el anchor point se ubica en el punto de origen, el elemento rotará sobre su mismo punto de origen.

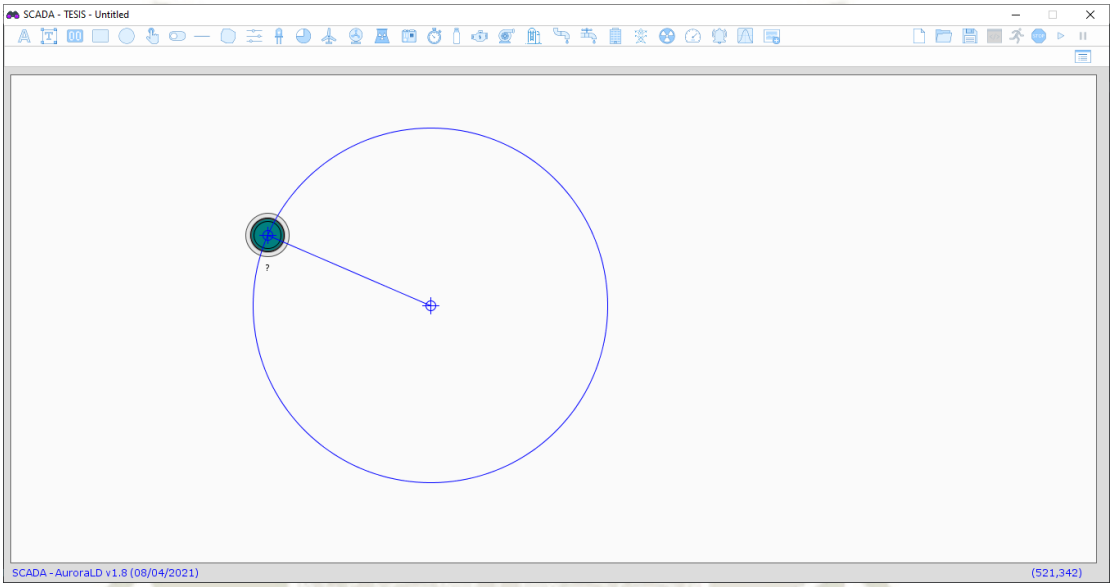


Figura 307: Previsualización de animación rotar

3.22.5. Color

La animación de color no se encuentra disponible para todos los elementos del entorno SCADA. Esta animación permite modificar el color de un determinado elemento al cumplirse las condiciones digitales o analógicas.

En el caso de variables digitales solo se puede escoger un color de encendido y un color de apagado, sin embargo, cuando se trabaja con variables analógicas es posible escoger como máximo 4 colores para 4 intervalos con sus respectivos mínimos y máximos. En la Figura 308 se muestra la ventana de parámetros de la animación color con la configuración necesaria para trabajar con tags de tipo analógico.

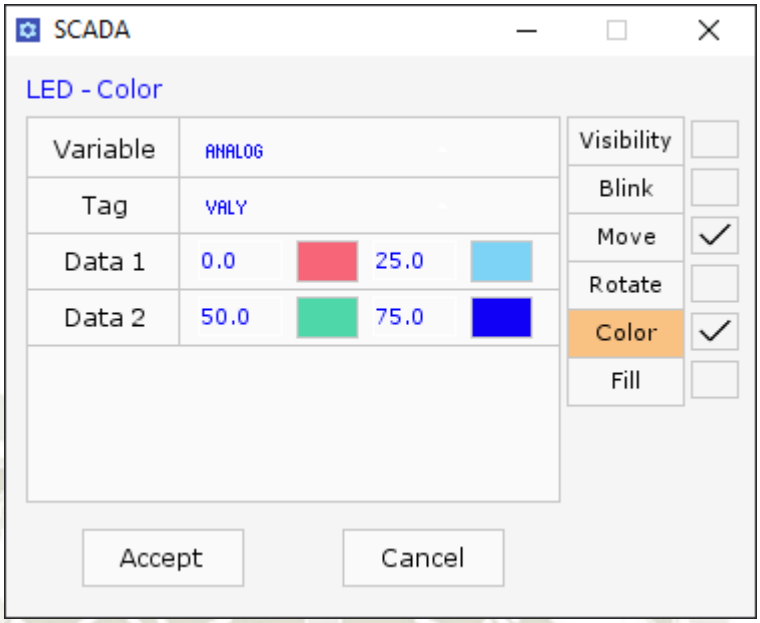


Figura 308: Animación color

El código que ejecuta la animación se encuentra en el interior de la clase de los elementos SCADA y comprende las siguientes variables principales:

- **row_cl:** Indica la fila en la tabla de datos del programa del tag asociado a la animación.
- **tag_cl:** Variable de tipo string que hace referencia al tag asociado a la animación.
- **var_cl:** El tipo de variable puede ser analógica o digital.
- **cn1_cl:** Color RGB del primer intervalo.
- **cn2_cl:** Color RGB del segundo intervalo.
- **cn3_cl:** Color RGB del tercer intervalo.
- **cn4_cl:** Color RGB del cuarto intervalo.
- **iv1_cl:** Máximo valor del primer intervalo.
- **iv2_cl:** Máximo valor del segundo intervalo.
- **iv3_cl:** Máximo valor del tercer intervalo.
- **iv4_cl:** Máximo valor del cuarto intervalo.

- **col_sel:** Variable asociada al color del elemento en un instante determinado. Esta variable cambia sus parámetros de acuerdo a las condiciones de la animación permitiendo así el cambio de color en los elementos.

```
//COLOR
int row_cl = -1;
String tag_cl;
String var_cl;
color cn1_cl;
color cn2_cl;
color cn3_cl;
color cn4_cl;
float iv1_cl;
float iv2_cl;
float iv3_cl;
float iv4_cl;
color col_sel;
```

Figura 309: Variables de la animación color

En la Figura 310 se muestra un ejemplo de la animación color asociado a un tag analógico. Se muestran distintos colores para el fluido en función del nivel del tanque:

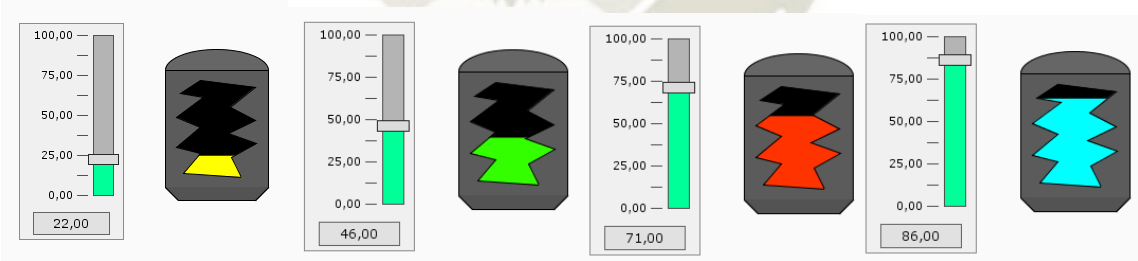


Figura 310: Ejemplo de animación color

3.22.6. Llenado (Fill)

Finalmente, la animación de llenado permite simular el llenado de una figura geométrica con un color elegido por el usuario. Esta animación solo se puede asociar a Tags analógicos. Cabe destacar que esta animación no está disponible para todos los elementos SCADA.

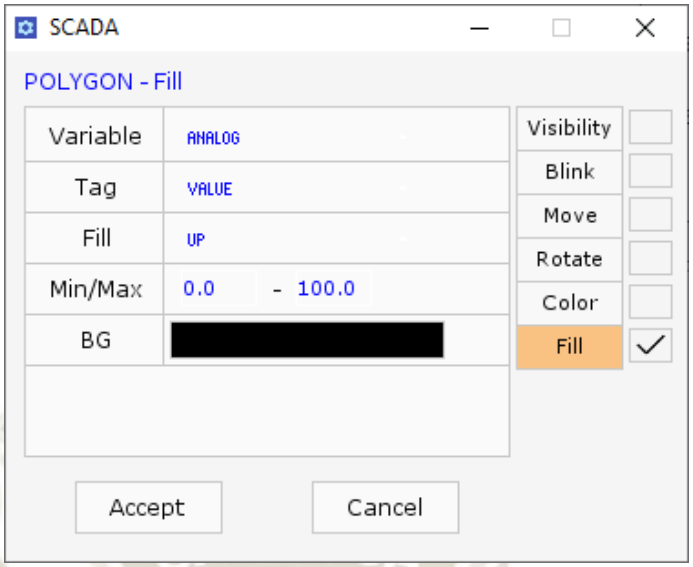


Figura 311: Animación llenado

El código que ejecuta la animación se encuentra en el interior de la clase de los elementos SCADA y comprende las siguientes variables principales:

- **row_fl:** Indica la fila del tag en la tabla de datos del programa asociada a la animación llenado.
- **tag_fl:** Variable de tipo string que hace referencia al tag asociado con la animación llenado.
- **type_fl:** En la animación llenado los tipos hacen referencia a la dirección de llenado. La dirección puede ser hacia arriba, hacia abajo, de izquierda a derecha o de derecha a izquierda.
- **min_fl:** Mínimo valor que puede asumir el tag.
- **max_fl:** Máximo valor que puede asumir el tag.
- **bg_fl:** Color RGB del fondo del elemento.
- **up_y:** Variable float que almacena el llenado actual en función del rango mínimo y máximo especificado para la dirección de abajo hacia arriba.

- **down_y:** Variable float que almacena el llenado actual en función del rango mínimo y máximo especificado para la dirección de arriba hacia abajo.
- **right_x:** Variable float que almacena el llenado actual en función del rango mínimo y máximo especificado para la dirección de izquierda a derecha.
- **left_x:** Variable float que almacena el llenado actual en función del rango mínimo y máximo especificado para la dirección de derecha a izquierda.

```
//FILL
int row_fl = -1;
String tag_fl;
String type_fl;
float min_fl;
float max_fl;
color bg_fl;
float up_y = 0;
float down_y = 0;
float right_x = 0;
float left_x = 0;
```

Figura 312: Variables de la animación llenado

En la Figura 313 se muestra un ejemplo de la animación llenado asociado a un tag analógico. Se muestran las distintas direcciones de llenado que se pueden escoger, siendo estas de izquierda a derecha en la figura: hacia arriba, hacia abajo, hacia la derecha y hacia la izquierda.

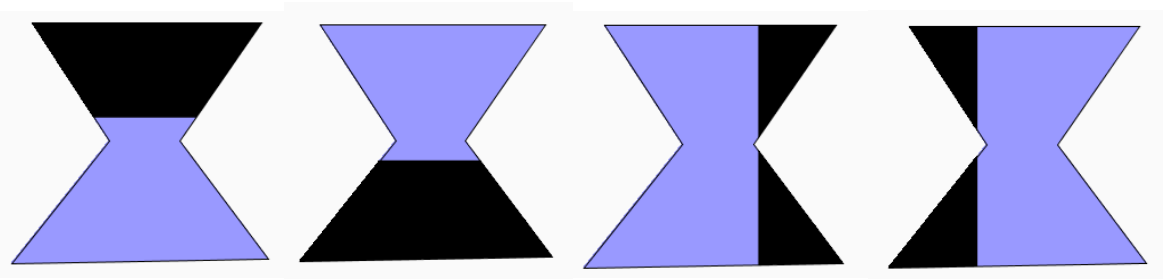


Figura 313: Ejemplo animación llenado

3.23. Eliminar elementos SCADA

Los elementos SCADA se eliminan haciendo nulos los objetos creados. Siempre que se quiera borrar un elemento se realiza el siguiente algoritmo:

- Se identifica el elemento que se desea borrar. Para esto es necesario conocer su número identificador y en que capa se encuentra dibujado.
- Todos los parámetros de los objetos siguientes al que se desea borrar son pasados a los objetos previos uno a uno en un bucle for, tal como se muestra en la Figura 314.
- El último elemento es eliminado haciendo al objeto nulo.

```
if (num!=hcool_counter-1) {
  for (int i = num+1; i<hcool_counter; i++) {
    //PARAMETERS
    HCOOL[i-1].x = HCOOL[i].x;
    HCOOL[i-1].y = HCOOL[i].y;
    HCOOL[i-1].col_cool = HCOOL[i].col_cool;
    HCOOL[i-1].layer = HCOOL[i].layer;
    HCOOL[i-1].cool_size = HCOOL[i].cool_size;
```

Figura 314: Ejemplo de código implementado para eliminar el elemento torre de enfriamiento

- La variable Layer (capa) es una variable global y empleada por todos los elementos SCADA. Esta variable debe cambiar su valor en el resto de elementos en base al eliminado.
- Todos aquellos objetos cuyo número de capa sea superior al elemento eliminado debe restarse uno para así mantener el número adecuado ante la falta de un elemento. Esto se realiza de forma automática empleando la función morph mostrada en la Figura 315.

```
void morph(int tracer, String action) {
  int ty = 0;
  for (int i=0; i<htext_counter; i++) {
    if (action.equals("Back")) {
      if (HTEXT[i].layer<tracer) {
        ty=HTEXT[i].layer+1;
        HTEXT[i].layer=ty;
      }
    } else if (action.equals("Front")) {
      if (HTEXT[i].layer>tracer) {
        ty=HTEXT[i].layer-1;
        HTEXT[i].layer=ty;
      }
    }
  }
```

Figura 315: Ejemplo de código para reajustar el número de capa del elemento texto

Lo mostrado en el código de la Figura 315 se efectúa en un bucle for para cada uno de los elementos siempre y cuando existan en el entorno SCADA. Es importante mencionar que la función morph también permite cambiar la capa de un elemento por la superior o inferior, colocando al elemento delante o detrás del resto de elementos.

3.24. Guardar y cargar un SCADA

En el entorno SCADA se tienen opciones para guardar, cargar y crear un nuevo Sistema de Supervisión y Adquisición de Datos (SCADA). Tanto para guardar como para cargar se utiliza un JFileChooser para poder escoger el directorio y el nombre del archivo tal como se muestra en la Figura 316.

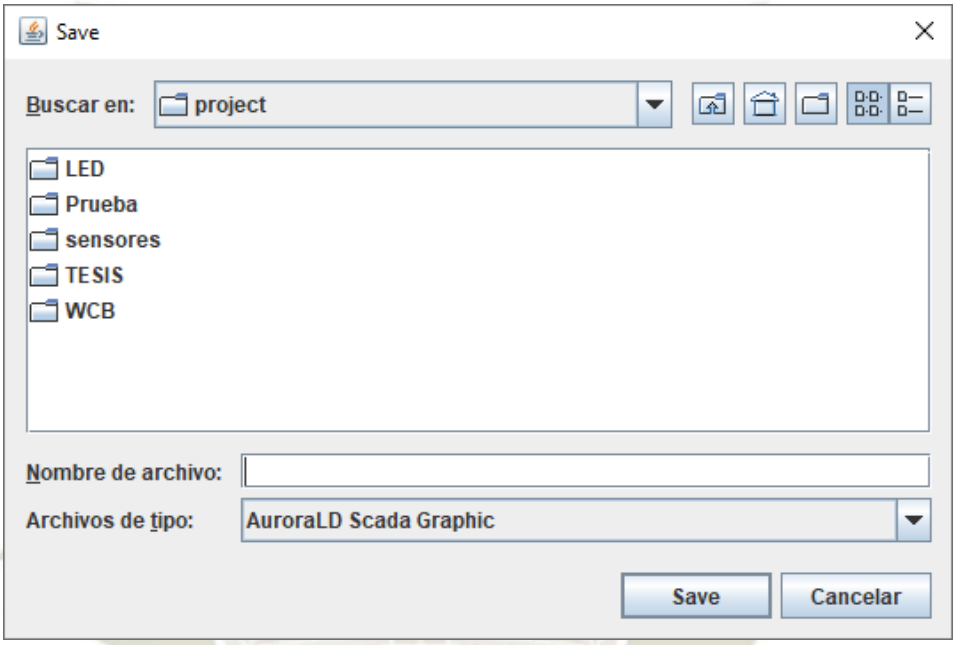


Figura 316: JFileChooser en el entorno SCADA

Los distintos elementos se guardan con sus respectivos parámetros y animaciones empleando un algoritmo que recopila toda la información esencial de cada uno de estos elementos. La información recopilada se convierte a un formato String y se procede a concatenar estos datos de manera que puedan ser almacenados en un archivo de texto (.txt). Un ejemplo del algoritmo en mención se muestra en la Figura 317 para el elemento alarma. En el resto de elementos se procede de la misma forma.

```
//SAVE ALARM PARAMETERS
String alarm_param[] = new String[halarm_counter*44];
int contador = 0;
for (int i=0; i<halarm_counter; i++) {
  //PARAMETERS
  alarm_param[contador]=str(HALARM[i].x);
  contador++;
  alarm_param[contador]=str(HALARM[i].y);
  contador++;
  alarm_param[contador]=str(HALARM[i].layer);
  contador++;
  alarm_param[contador]=str(HALARM[i].num);
  contador++;
  alarm_param[contador]=str(HALARM[i].row_tag);
  contador++;
  alarm_param[contador]=str(HALARM[i].alarm_sound);
  contador++;
  alarm_param[contador]=str(HALARM[i].alarm_action);
  contador++;
  alarm_param[contador]=HALARM[i].tag;
  contador++;
  alarm_param[contador]=str(HALARM[i].alarm_type);
```

Figura 317: Ejemplo de código para guardar parámetros del elemento alarma

En AuroraLD Studio se ha definido una extensión distinta a .txt para los archivos que corresponden al entorno SCADA. La extensión escogida es Programmable Control System (.pcs). Processing admite colocar cualquier extensión en los archivos ya que básicamente guarda los datos como texto pudiendo incluso visualizar el archivo guardado empleando un editor de texto.

Por otro lado, para cargar los datos del archivo guardado al entorno SCADA se implementa un algoritmo que busca línea a línea los datos y los agrega en la función del constructor de cada una de las clases para crear los objetos y poder mostrarlos. El proceso

es bastante directo puesto que los datos se guardan en orden específico y conocido, por ende, cargar los datos del archivo guardado no supone un problema. En la Figura 318 se muestra el constructor de la clase HMI_FAN a la que se está asociando datos cargados para poder crear los objetos respectivos.

```
for (int i=0; i<hfan_counter; i++) {
  contador++;
  HFAN[i] = new HMI_FAN(int(pcs_file[contador]), int(pcs_fil
    color(unhex(pcs_file[contador+3])), int(pcs_file[contadc
    pcs_file[contador+7], int(pcs_file[contador+8]),
    pcs_file[contador+9], int(pcs_file[contador+10]), float(
    float(pcs_file[contador+12]), pcs_file[contador+13], pcs
    int(pcs_file[contador+16]), float(pcs_file[contador+17])
    pcs_file[contador+19], pcs_file[contador+20], int(pcs_fi
    pcs_file[contador+23], int(pcs_file[contador+24]), float
    pcs_file[contador+27], pcs_file[contador+28], int(pcs_fi
    pcs_file[contador+31], int(pcs_file[contador+32]), float
    pcs_file[contador+35], pcs_file[contador+36], float(pcs_
    int(pcs_file[contador+39]), int(pcs_file[contador+40]),
```

Figura 318: Ejemplo de código para cargar el elemento ventilador (Fan)

Finalmente, para la opción de Nuevo se ha implementado un algoritmo que destruye los objetos que hayan sido creados haciendo que cada uno de los objetos regresen a un estado Nulo (Null). Este código es similar al empleado para eliminar los elementos SCADA (mostrado en un apartado anterior). En la Figura 319 observamos que mediante el uso de bucles for se llama a cada uno de los elementos presentes en el entorno SCADA y se asigna el valor nulo.

```
void hmi_clean() {
  for (int i=0; i<htext_counter; i++) {
    HTEXT[i]=null;
  }

  for (int i=0; i<hin_counter; i++) {
    HIN[i]=null;
  }

  for (int i=0; i<hout_counter; i++) {
    HOUT[i]=null;
  }
```

Figura 319: Función hmi_clean para crear un nuevo espacio de trabajo SCADA

3.25. Implementacion del banco de pruebas

El banco de pruebas que se implementará se denomina PLC1805 y está basado en Arduino Nano. El PLC en mención no es más que un módulo de entradas y salidas con indicadores LEDs. El módulo de pruebas consta de 6 entradas y 6 salidas digitales, así como también 4 entradas y 4 salidas analógicas. La alimentación del módulo se realiza a través del pin VIN o mediante el cable USB. Con la finalidad de probar las entradas digitales se ha incorporado en el módulo pulsadores.

El banco de pruebas que se muestra en la Figura 320 no es necesario para establecer la comunicación con el software, sin embargo, se recomienda implementar algo similar para observar el cambio en las entradas y salidas digitales. Por otro lado, las entradas y salidas analógicas se pueden monitorear en el software tanto si se usa la tabla de datos del programa o por medio de un sistema SCADA.

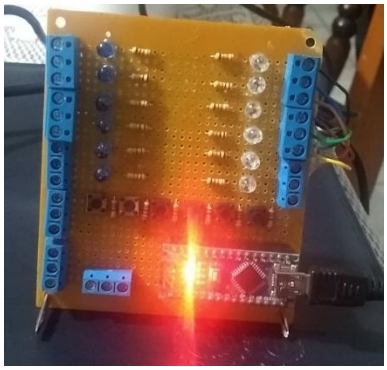


Figura 320: Banco de pruebas PLC1805

Incluso es posible modificar y acondicionar las entradas y/o salidas digitales para trabajar en un rango más industrial ya sea a 10 o 24 V. Por el momento el módulo trabaja solo a nivel de 5V tanto para entradas y para salidas.

Los componentes empleados para el desarrollo del módulo se muestran en la Tabla 11. También se especifican los costos de los componentes en la tabla con la finalidad de mostrar que se puede armar un módulo como el mostrado en la Figura 320 por un costo bajo de 35.4 soles.

Tabla 11: Componentes módulo PLC1805

Componente	**Valor**	**Cantidad**	**Costo**
Leds	Azul 5mm	6	s/. 1.20
Leds	Verde 5mm	6	s/. 1.20
Pulsadores	4 pines	6	s/. 3.00
Arduino	Nano	1	s/. 15.00
Bornera	3 pines	9	s/. 9.00
Resistencias	220 ohms	18	s/. 3.00
Placa para soldar	85 x 95 mm	1	s/. 3.00
TOTAL			**s/. 35.40**

El circuito a implementar se muestra en la Figura 321.

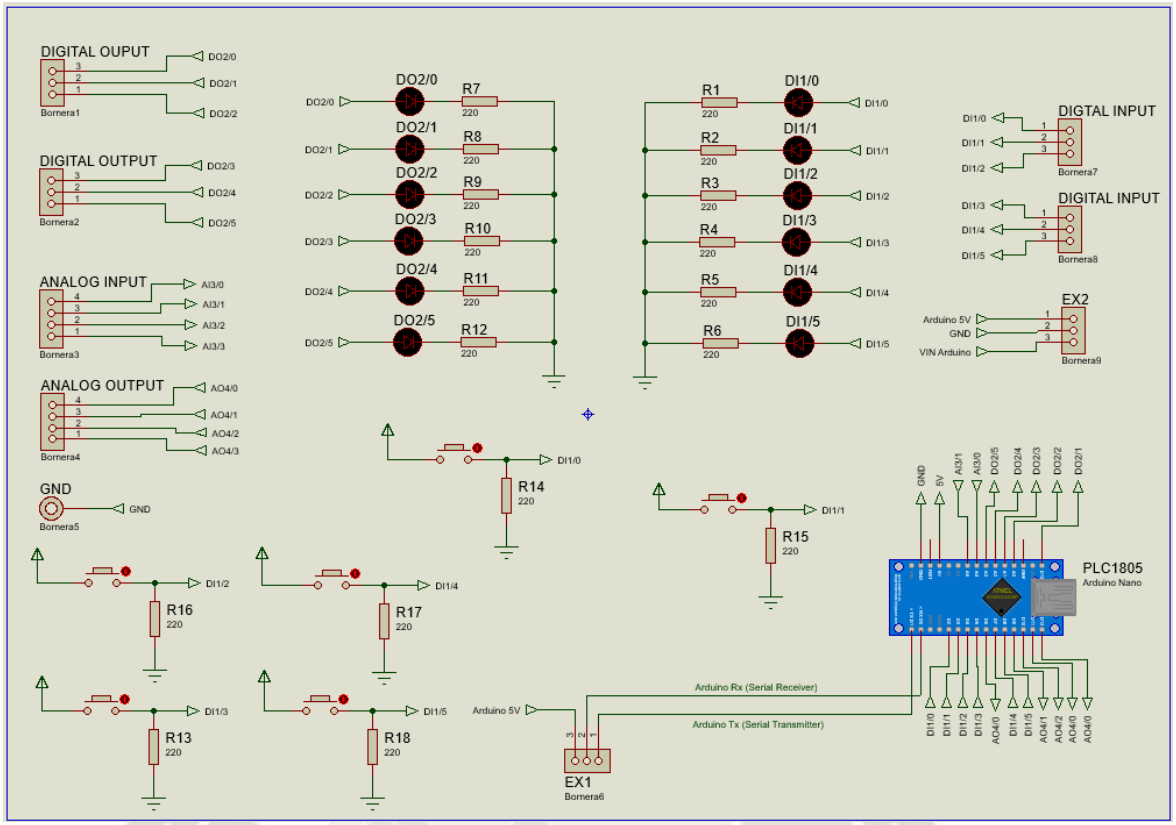


Figura 321: Circuito general del PLC1805

3.25.1. Arduino Nano

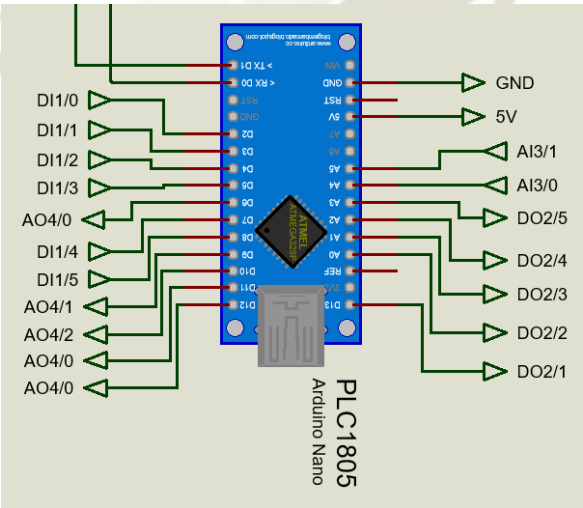


Figura 322: Conexiones en Arduino

Las conexiones especificadas en la Figura 332 deben de respetarse puesto que así es como AuroraLD Studio entenderá y definirá las entradas y/o salidas analógicas/digitales.

El esquema de Arduino que se emplea en Proteus no permite crear conexiones con los pines analógicos A6 y A7, pero estos pines están también definidos como entradas analógicas en AuroraLD Studio siendo A6 y A7 las entradas analógicas AI:3/2 y AI:3/3 respectivamente.

Esta información con las conexiones de Arduino y las direcciones correspondientes en el software están también detalladas en las propiedades del controlador, opción que se encuentra en VIEW->Controller properties (Figura 323).

Arduino Nano Pins

Pin	Address	Description
Pin 2	DI:1/0	Digital Input 1
Pin 3	DI:1/1	Digital Input 2
Pin 4	DI:1/2	Digital Input 3
Pin 5	DI:1/3	Digital Input 4
Pin 7	DI:1/4	Digital Input 5
Pin 8	DI:1/5	Digital Input 6
Pin 12	DO:2/0	Digital Output 1
Pin 13	DO:2/1	Digital Output 2
Pin 14	DO:2/2	Digital Output 3
Pin 15	DO:2/3	Digital Output 4
Pin 16	DO:2/4	Digital Output 5
Pin 17	DO:2/5	Digital Output 6
Pin A4	AI:3/0	Analog Input 1
Pin A5	AI:3/1	Analog Input 2
Pin A6	AI:3/2	Analog Input 3
Pin A7	AI:3/3	Analog Input 4
Pin 6	AO:4/0	Analog Output 1
Pin 9	AO:4/1	Analog Output 2
Pin 10	AO:4/2	Analog Output 3
Pin 11	AO:4/3	Analog Output 4

Figura 323: PLC1805 (Controller properties)

3.25.2. Entradas Digitales

Las entradas digitales se componen básicamente de 2 borneras de 3 pines como las mostradas en la Figura 324.

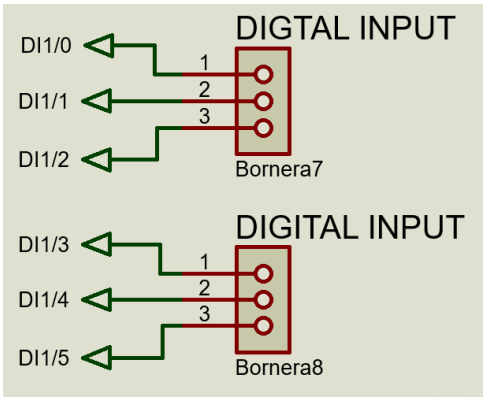


Figura 324: Borneras entradas digitales

Cuando ingresa una señal al módulo esta señal se reparte en dos: una va directamente a un pin de entrada en Arduino y la otra va a uno de los leds que tiene asignado. En la Figura 325 se muestra la conexión de los indicadores de entrada.

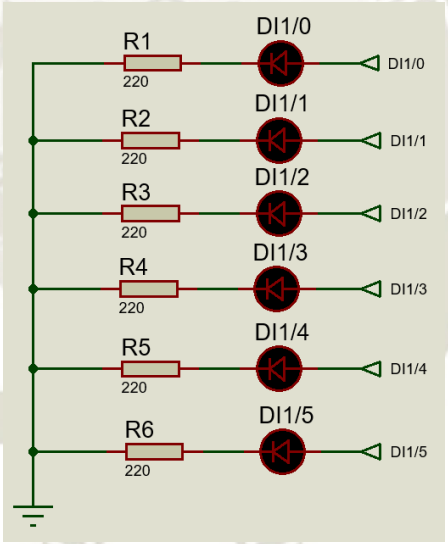


Figura 325: LEDs indicadores de entradas digitales

Se comprueba el funcionamiento de las entradas digitales y su activación respectiva empleando los pulsadores que se han agregado al circuito. Al presionar un pulsador uno de los LEDs que tiene asociado debe de encender.

Asimismo, en el software debe notarse un cambio en esa entrada al pasar de un estado 0 lógico (0 voltios) a un estado 1 lógico (5 voltios).

3.25.3. Salidas Digitales

Las salidas digitales se componen básicamente de 2 borneras de 3 pines como se muestra en la Figura 326.

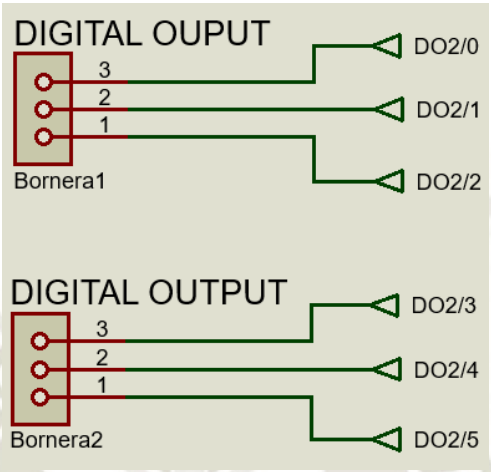


Figura 326: Borneras salidas digitales

Es importante recordar que la tensión de salida es solamente de 5V por lo que si se requieren emplear dispositivos de mayor tensión es necesario emplear un relé o un transistor que permita el control de esos dispositivos.

Cuando se activa una de las salidas de Arduino esta se transmite directamente a las borneras de salida y a los LEDs asociados.

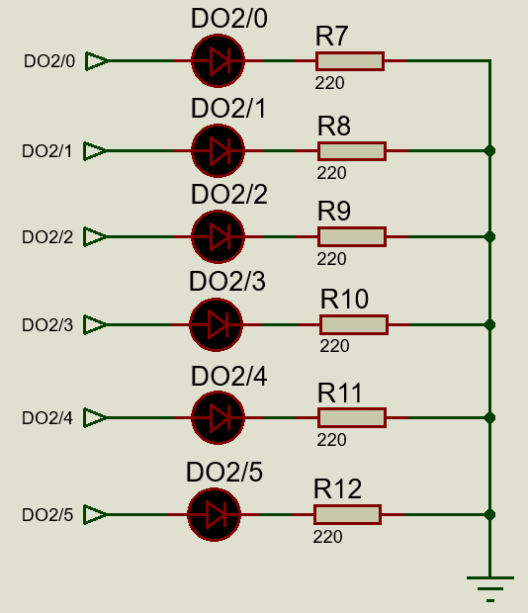


Figura 327: Indicadores de salidas digitales

3.25.4. Entradas Analógicas

El PLC1805 solo cuenta con 4 entradas analógicas las cuales van directamente conectadas de las borneras a los pines de Arduino correspondientes tal como se muestra en la Figura 328.

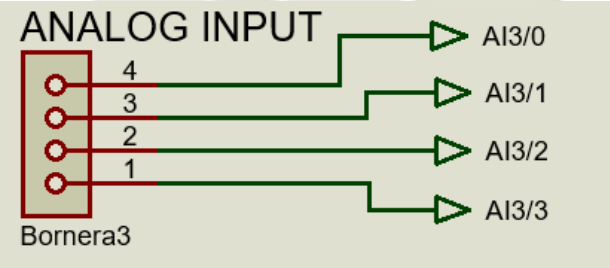


Figura 328: Entradas Analógicas borneras

Las direcciones de las entradas analógicas llevan las letras AI (Analog Input). Estas entradas van conectadas al conversor analógico/digital de Arduino el cual posee una resolución de 10 bits permitiendo así tener lecturas entre 0 (0 voltios) y 1023 (5 voltios).

3.25.5. Salidas Analógicas

El módulo implementado solo cuenta con 4 salidas analógicas que van conectadas directamente desde las borneras hasta los pines PWM en Arduino tal como se muestra en la Figura 329. Es importante recordar que la salida PWM solo es de 8 bits por lo que solo se le puede pedir a Arduino valores entre 0 a 255 para la salida analógica.

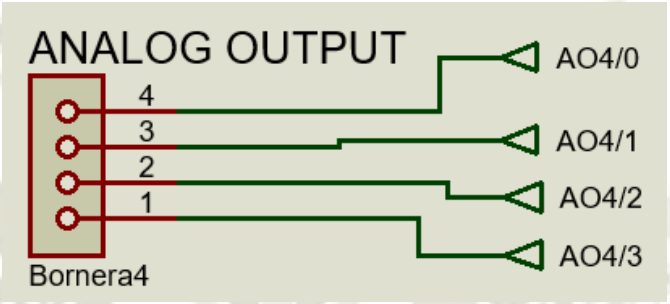


Figura 329: Salidas analógicas borneras

Las direcciones de las salidas analógicas van representadas por las letras AO (Analog Output).

3.25.6. Borneras Adicionales

El módulo también cuenta con dos borneras adicionales que permiten la conexión de los pines Rx (Receptor Serial) y Tx (Transmisión Serial) para poder conectar algún dispositivo si es que se desea. Asimismo, otra de las borneras da acceso a los pines de alimentación y tierra respectivamente para los diferentes sensores o cargas que se conecten al dispositivo.

3.25.7. Pulsadores

Finalmente, para poder probar el módulo con mayor facilidad se opta por incluir pulsadores. La implementación de estos pulsadores no es necesaria sin embargo ayuda a realizar las pruebas sin requerir la conexión de equipos externos que brinden una señal digital de entrada al PLC1805.

En el módulo se han implementado 6 pulsadores correspondiente a cada una de las entradas digitales. El circuito mostrado en la Figura 339 se emplea para cada uno de los pulsadores.

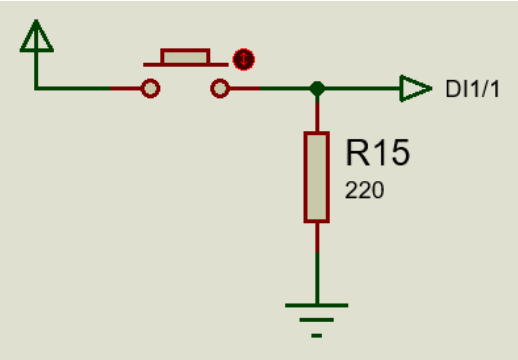


Figura 330: Pulsadores (entrada digital)

La salida del circuito mostrado en la Figura 339 se conecta directamente en las borneras correspondientes a las entradas digitales.

Luego de implementar el módulo y construir un case para el mismo se obtienen los resultados mostrados en la Figura 340. En esta figura se puede observar el módulo con todas las partes especificadas en la imagen del lado izquierdo. Por otro lado, en la imagen del lado derecho se muestra el funcionamiento de una entrada digital.

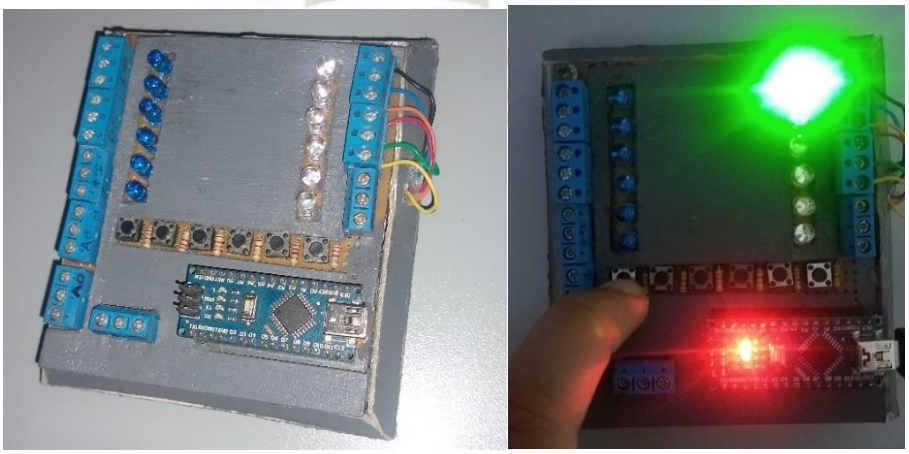


Figura 331: PLC1805 Implementado

CAPÍTULO IV

PRUEBAS Y RESULTADOS

CAPÍTULO IV – PRUEBAS Y RESULTADOS

4.1. Cumplimiento del estándar IEC 61131-3

El entorno de programación de AuroraLD Studio posee una interfaz intuitiva de fácil uso, siendo el software de programación Ladder de los controladores PLC1805 y PLC1607 para uso académico de bajo costo. Las instrucciones empleadas son similares al software RSLogix 5000 de Rockwell Automation.

IEC ha desarrollado especificaciones respecto a la programación de PLCs en un intento por unificar los equipos y lenguajes de programación que se deben emplear en la industria. El estándar IEC 61131-3 es una guía para programación de PLCs y se espera que los fabricantes cumplan con parte y no todo el estándar.

Las especificaciones establecidas por el estándar IEC 61131-3 para la programación de PLCs y su cumplimiento con el software AuroraLD Studio son evaluadas en este apartado. La evaluación está más enfocada en los lenguajes de programación de PLCs especificados en la parte 3 del estándar IEC 61131.

Este apartado se construye en base al estándar IEC 61131-3 [11] y a un manual de cumplimiento del estándar realizado por Rockwell Automation para el software de los controladores Logix 5000 [23] adaptando algunas de las tablas encontradas en esta fuente.

4.1.1. Tags y tipos de datos

El estándar especifica que las direcciones en un entorno de programación se reemplazan por variables con un nombre y tipo de dato específico. Asignar un tipo de dato evita errores al momento de ejecutar el programa. Los nombres de las variables “Tags” o también denominados etiquetas se declaran en los Tags del programa y estos Tags pueden contener números (0-9), letras (A-Z) y guion bajo (“_”). El estándar especifica un mínimo

de 6 caracteres para los Tags, sin embargo, AuroraLD Studio puede trabajar con un carácter como mínimo. Asimismo, el estándar especifica que no hay diferencia entre mayúsculas y minúsculas para los Tags, sin embargo, en AuroraLD Studio se hace esta distinción ya que la declaración de variables es sensible a mayúsculas y minúsculas por lo tanto no se cumple con esta parte del estándar.

Además, en AuroraLD Studio los datos de las variables son globales en muchos casos y en otros están simplemente restringidos a una determina instrucción tal cual lo especifica el estándar. Estos datos se comparten entre diferentes funciones y bloques de funciones al escribir y leer datos directamente en la tabla de valores la cual se puede encontrar en Program Tags.

El estándar IEC 61131-3 define también los tipos de datos básicos a usar en el software de programación. En la Tabla 12 se muestran los tipos de datos que soporta AuroraLD Studio (No todos los tipos de datos mostrados en la tabla son los únicos especificados en el estándar).

Tabla 12: Tipos de datos soportados por AuroraLD Studio

Tipo de Dato	**Descripción**	**Bits**	**Rango**	**Valor Inicial**
BOOL	Booleano	1	0/1 o False/True	0
DINT	Doble entero	32	$-2^{31}/2^{31}-1$	0
REAL	Números Reales	32 (4 bytes – La cantidad no hace referencia a lo establecido en el estándar)	$-3.4^{38}/3.4^{38}$	0

AuroraLD Studio cuenta con otros 3 tipos de variables necesarias que identifican a bloques de funciones básicos en el software. Estas variables son necesarias para generar otras variables internas correspondientes a cada uno de los bloques de funciones:

Tabla 13: Tipos de variables adicionales

Tipo de Dato	Descripción	Bits
TIMER	Tipo de dato para temporizadores (Se generan variables DINT y BOOL a partir de esta variable)	DINT (32) / BOOL (1)
COUNTER	Tipo de dato para contadores (Se generan variables DINT y BOOL a partir de esta variable)	DINT (32) / BOOL (1)
PID	Tipo de dato para un bloque de control PID (Se generan variables REAL a partir de esta variable)	$-3.4^{38}/3.4^{38}$

4.1.2. Lenguajes de programación

En el estándar IEC 61131-3 se detallan 6 lenguajes de programación y por lo menos uno de los mencionados debe ser utilizado por el software de programación para poder satisfacer el cumplimiento del estándar. Los lenguajes de programación detallados hacen referencia a 3 lenguajes textuales y 3 lenguajes gráficos.

Textuales

- Lista de instrucciones (IL)
- Texto estructurado (ST)
- Gráfico de función secuencial (versión textual)

Gráficos

- Diagrama Ladder (LD)
- Diagrama de bloque de funciones (FBD)
- Gráfico de función secuencial (versión gráfica)

AuroraLD Studio soporta uno de los lenguajes gráficos mencionados en el estándar IEC 61131-3. El lenguaje o diagrama Ladder es el único lenguaje de programación implementado en el software AuroraLD Studio.

Debido a que los PLC1805 y PLC1607 están basados en las placas Arduino también existe la posibilidad de usar el entorno de programación Arduino IDE. Los programas creados en AuroraLD Studio usan el entorno de Arduino IDE para poder ser cargados en el microcontrolador. Existe un proceso de conversión del diagrama Ladder a código entendible por la plataforma Arduino incluido en el mismo software.

4.1.3. Instrucciones

El estándar IEC 61131-3 define los siguientes 8 grupos de funciones básicas:

- Funciones de conversión de tipos de datos
- Funciones numéricas
- Funciones aritméticas
- Funciones bit-string (bit-shift)
- Funciones de selección y comparación
- Funciones de string (character)
- Funciones para datos de tipo tiempo
- Funciones para datos de tipo enumerados

AuroraLD Studio posee algunas de las funciones numéricas, aritméticas, selección y comparación especificadas en el estándar IEC 61131-3. A continuación se detallan que funciones especificadas en el estándar se incluyen en el software:

Tabla 14: Funciones Numéricas

Función	Tipo de Dato	Descripción
ABS	REAL/DINT	Calcula el valor absoluto de un número
SQRT	REAL/DINT	Calcula la raíz cuadrada de un número
SIN	REAL/DINT	Calcula el seno de un número
COS	REAL/DINT	Calcula el coseno de un número
TAN	REAL/DINT	Calcula la tangente de un número
ASIN	REAL/DINT	Calcula el arco seno de un número
ACOS	REAL/DINT	Calcula el arco coseno de un número
ATAN	REAL/DINT	Calcula el arco tangente de un número

Tabla 15: Funciones Aritméticas

Función	Tipo de Dato	Descripción
ADD	REAL/DINT	Calcula la suma de dos números
MUL	REAL/DINT	Calcula la multiplicación de dos números
SUB	REAL/DINT	Calcula la resta de dos números
DIV	REAL/DINT	Calcula la división de dos números
MOD	REAL/DINT	Calcula el residuo de una división
MOV	REAL/DINT	Asigna un valor a una variable determinada

Tabla 16: Función de Selección

Función	Tipo de Dato	Descripción
LIM	REAL/DINT	Evalúa un determinado valor para saber si se encuentra dentro de un límite especificado por un valor mínimo y máximo

Tabla 17: Funciones de Comparación

Función	Tipo de Dato	Descripción
GRT	REAL/DINT	Mayor que
GEQ	REAL/DINT	Mayor o igual que
EQU	REAL/DINT	Igual que
LES	REAL/DINT	Menor que
LEQ	REAL/DINT	Menor o igual que
NEQ	REAL/DINT	No es igual que

El estándar IEC 61131-3 define también 5 grupos de bloques de funciones básicos:

- Elementos biestables
- Edge detection
- Contadores
- Temporizadores
- Bloques de función para comunicación

AuroraLD Studio posee algunos de los bloques de funciones: contadores y temporizadores especificados en el estándar IEC 61131-3. A continuación se detallan que bloques de funciones especificados en el estándar se incluyen en el software:

Tabla 18: Contadores

Bloque de función	Variables de Salida	Descripción
CTU	EN, DN, ACC, PRE	Contador ascendente
CTD	EN, DN, ACC, PRE	Contador descendente

Tabla 19: Temporizadores

Bloque de función	Variables de Salida	Descripción
TON	EN, DN, ACC, PRE	Temporizador on delay
TOF	EN, DN, ACC, PRE	Temporizadores off delay
RTO	EN, DN, ACC, PRE	Temporizador Retentivo

AuroraLD Studio define otras instrucciones (funciones y bloques de funciones) que se consideran necesarias pese a no estar estipuladas en el estándar. Estas instrucciones se explican con mayor detalle en el manual del entorno Ladder correspondiente. El estándar brinda completa libertad para la integración de nuevas instrucciones ya que la lista especificada en el estándar se puede considerar como opcional, pero se espera que los softwares de programación cuenten como mínimo con dichas instrucciones al considerarse básicas.

4.1.4. Portabilidad del software

El software desarrollado no tiene integración con fabricantes de otras marcas de PLCs tanto a nivel de hardware como a nivel de software. El software tiene un propósito académico y solo funciona con la plataforma Arduino siempre y cuando posea el programa del PLC1805 o PLC1607 según corresponda. Según el estándar IEC 61131-3 se requiere que los programas que cumplan con el estándar posean integración con diferentes fabricantes, sin embargo, el estándar no especifica un formato determinado para los archivos.

AuroraLD Studio trabaja con 3 formatos de archivo diferente:

- Formato .pl – Contiene la información de los Tags, Trends y diagrama Ladder de un proyecto creado en AuroraLD Studio.

- Formato .ara – Contiene los parámetros básicos de todo proyecto creado en AuroraLD Studio.
- Formato .pcs – Contiene los gráficos creados y la información del sistema SCADA.

4.1.5. Obtención de datos

AuroraLD Studio colecta los valores del PLC de forma síncrona, es decir, los valores son recolectados al terminar un ciclo del diagrama Ladder. Cuando el ciclo termina los valores se escriben en las salidas y se actualizan los datos de entrada para iniciar un nuevo ciclo. Esta información está detallada en el estándar IEC 61131-3 respecto a la ejecución y comunicación de datos del PLC al software de programación.

4.1.6. Diagrama Ladder

AuroraLD Studio cumple con el llamado de funciones y bloques de funciones respectivamente. Según el estándar estas funciones o bloques de funciones deben representarse por cajas rectangulares y pueden tener uno o más parámetros de entrada o salida. Estos parámetros pueden ser de cualquier tipo, pero por lo menos uno de ellos debe ser de tipo Booleano y debe tener conexión directa o indirecta con los rieles de energía (Power Rails) izquierdo o derecho.

4.1.7. Tablas de cumplimiento IEC 61131-3

A continuación, solo se muestran las tablas en las que el software AuroraLD Studio cumple con lo estipulado en el estándar IEC 61131-3.

Tabla 20: Identificadores (Identifiers)

Característica	Descripción
Mayúsculas letras y números	AuroraLD Studio permite el uso de mayúsculas y números en cada caja de texto presente en el software
Minúsculas letras, números y guion bajo	Está permitido el uso de minúsculas, números y guiones bajos en cada una de las cajas de texto presentes en el software

Tabla 21: Literales Numéricos (Numeric Literals)

Característica	Descripción
Literal entero	Hace referencia a los valores enteros empleados en AuroraLD Studio. Por ejemplo: 16, 7, 0, -24, 17, -11, 19, etc.
Literal real	Hace referencia a los valores reales empleados en AuroraLD Studio. Por ejemplo: -16.07, 9.17, 19.05, -8.11, 3.20, etc.
Literal booleano	Hace referencia a los valores booleanos empleados en AuroraLD Studio. Siendo estos valores falso o verdadero, 0 o 1, etc.

Tabla 22: Tipos de datos elementales (Elementary data types)

Característica	Descripción
Tipo de dato BOOL	Dependiendo de la elección del usuario las variables pueden ser de tipo booleano. Se definen en AuroraLD Studio bajo el nombre BOOL.
Tipo de dato DINT	Dependiendo de la elección del usuario las variables pueden ser de tipo entero. Se

	definen en AuroraLD Studio bajo el nombre INT pese a ser un doble entero.
Tipo de dato REAL	Dependiendo de la elección del usuario las variables pueden ser del tipo real. Se definen en AuroraLD Studio bajo el nombre REAL.

Tabla 23: Declaración de variables

Característica	**Descripción**
Variables con tipo de dato elemental	AuroraLD Studio usa algunas de las variables especificadas en el estándar IEC 61131-3.

Tabla 24: Inicialización de variables

Característica	**Descripción**
Inicialización de variables con tipos de dato elemental	AuroraLD Studio usa algunas de las variables especificadas en el estándar IEC 61131-3 para su inicialización.

Tabla 25: Control de ejecución gráfico usando EN Y ENO

Característica	**Descripción**
Uso de EN y ENO	El uso de EN y ENO está implementado en AuroraLD Studio, sin embargo, no está descrito en ninguna de las instrucciones. Se aplica al evaluar y guardar los resultados booleanos de la parte izquierda (Saving) para posteriormente ejecutar las instrucciones en la parte derecha (Computing) del diagrama Ladder.

Tabla 26: Funciones escritas y sobrecargadas

Característica	Descripción
Funciones sobrecargadas	La mayoría de funciones en AuroraLD Studio no están sujetas a un único tipo de dato y pueden usar tanto variables INT como REAL.

Tabla 27: Funciones para conversión de tipos de dato

Característica	Descripción
Funciones de conversión TO	AuroraLD Studio permite la conversión de valores angulares, es decir, de radianes a grados sexagesimales y viceversa. Estas funciones están implementadas bajo el nombre RAD (DEG to RAD) y DEG (RAD to DEG).

Tabla 28: Funciones numéricas y aritméticas

Característica	Descripción
Valor absoluto	Permite calcular el valor absoluto de un número. Instrucción ABS.
Raíz cuadrada	Permite calcular la raíz cuadrada de un número. Instrucción SQR.
Logaritmo natural	Permite calcular el logaritmo natural de un número. Esta función solo está incluida dentro de la función “Compute” (CPT) empleando logE(a). Donde “a” es un número cualquiera.
Logaritmo base 10	Permite calcular el logaritmo en base 10 de un número. Esta función solo está incluida

	dentro de la función “Compute” (CPT) empleando log10(a). Donde “a” es un número cualquiera.
Seno	Permite calcular el seno de un número. Instrucción SIN.
Coseno	Permite calcular el coseno de un número. Instrucción COS.
Tangente	Permite calcular la tangente de un número. Instrucción TAN.
Arcoseno	Permite calcular el arcoseno de un número. Instrucción ASN.
Arcocoseno	Permite calcular el arcocoseno de un número. Instrucción ACS.
Arcotangente	Permite calcular el arcotangente de un número. Instrucción ATN.

Tabla 29: Funciones aritméticas

Característica	**Descripción**
Suma	Permite realizar una suma. Instrucción ADD.
Multiplicación	Permite realizar una multiplicación. Instrucción MUL.
Resta	Permite realizar una resta. Instrucción SUB.
División	Permite realizar una división. Instrucción DIV.
Exponenciación	Permite elevar un número a un especificado exponente. Disponible solo en la función “Compute” (CPT) empleando pow(a, b). Donde “a” es la base y “b” el exponente.
Mover	Permite asignar un valor a una variable determinada. Instrucción MOV.

Tabla 30: Funciones de selección

Característica	Descripción
Límite	Función que permite evaluar si un determinado valor se encuentra dentro de un rango máximo y mínimo. Instrucción LIM.

Tabla 31: Funciones de comparación

Característica	Descripción
Mayor que	Determina si un valor es mayor que otro valor. Instrucción GRT.
Mayor o igual que	Determina si un valor es mayor o igual que otro valor. Instrucción GEQ.
Igual que	Determina si un valor es igual que otro valor. Instrucción EQU.
Menor que	Determina si un valor es menor que otro valor. Instrucción LES.
Menor o igual que	Determina si un valor es menor o igual que otro valor. Instrucción LEQ.
No igual que	Determina si un valor no es igual que otro valor. Instrucción NEQ.

Tabla 32: Bloque de funciones estándar de contadores

Característica	Descripción
Contador ascendente	Permite realizar una cuenta ascendente. Instrucción CTU.
Contador descendente	Permite realizar una cuenta descendente. Instrucción CTD.

Tabla 33: Bloque de funciones estándar de temporizadores

Característica	Descripción

Temporizador On-delay	El temporizador está implementado en AuroraLD Studio bajo la instrucción TON.
Temporiador Off-delay	El temporizador está implementado en AuroraLD Studio bajo la instrucción TOF.
Temporizador RTO	El temporizador está implementado en AuroraLD Studio bajo la instrucción RTO.

Tabla 34: Declaración del programa

Característica	Descripción
Declaración del programa	Especificado en la rutina principal (Main Routine) en el software AuroraLD Studio.
Declaración de entradas	Se declaran las entradas del programa en los Program Tags o Etiquetas del programa.
Declaración de salidas	Se declaran las salidas del programa en los Program Tags o Etiquetas del programa.
Inicialización de entradas	Se inicializan las variables de entrada según lo elija el usuario (variables internas).
Inicialización de salidas	Se inicializan las variables de salida según lo elija el usuario (variables internas).

Tabla 35: Configuración y declaración de recursos

Característica	Descripción
Variables globales	Declaración de variables globales. Todas las variables especificadas en el programa AuroraLD Studio son variables globales accesibles por otras funciones o bloques de funciones dentro del programa.

Tabla 36: Carriles de alimentación y conexión de elementos

Característica	Descripción
Carril de alimentación izquierdo	El carril de alimentación izquierdo está presente en el entorno de programación Ladder.
Carril de alimentación derecho	El carril de alimentación derecho está presente en el entorno de programación Ladder.
Enlace horizontal (conexión)	Existen conexiones horizontales para la conexión de distintos elementos en el entorno de programación Ladder.
Enlace vertical (conexión)	Existen conexiones verticales para la conexión de distintos elementos en el entorno de programación Ladder.

Tabla 37: Contactos

Característica	Descripción
Contactos normalmente abiertos	Instrucción implementada en AuroraLD Studio bajo el nombre XIC.
Contactos normalmente cerrados	Instrucción implementada en AuroraLD Studio bajo el nombre XIO.

Tabla 38: Bobinas

Característica	Descripción
Bobina (Coil)	Instrucción implementada en AuroraLD Studio bajo el nombre OTE.
Latch (Set)	Instrucción implementada en AuroraLD Studio bajo el nombre OTL.

Unlatch (Reset)	Instrucción implementada en AuroraLD Studio bajo el nombre OTU.

Con la información especificada se puede concluir que el software AuroraLD Studio cumple parte del estándar IEC 61131-3. Como se mencionó antes las tablas presentadas en este apartado han sido adaptadas de [23].

4.2. Pruebas

Se realizaron distintas pruebas en el software y se comprobó el funcionamiento de cada una de las instrucciones, elementos, gráficos de tendencia, herramientas, etc. En los puntos siguientes se presenta el funcionamiento del software en los distintos entornos de trabajo mediante el desarrollo de un ejemplo.

Ejemplo

Se tiene un tanque de 60L de 1.5m de alto el cual tiene acoplado una bomba B1 que permite el llenado y una bomba B2 que permite la descarga del tanque. El nivel de líquido se mide a través de un sensor ultrasónico (S1). Adicionalmente, se cuenta con un panel de mando, el cual incluye 2 pulsadores: un pulsador (P1) de parada de emergencia y otro (P2) que permite el llenado de recipientes de 10L. El sistema también cuenta con un sensor capacitivo en el lugar de descarga el cual indica si el recipiente donde se descarga el líquido ha alcanzado 10L. Finalmente se tienen indicadores LED y un sensor que detecta cuando se ha colocado un recipiente en la zona de descarga. El funcionamiento del sistema debe ser el siguiente:

- Si el nivel de líquido cae a 10L (0.25m) la bomba B1 debe encenderse y llenar el tanque hasta 60L (1.5m). En este estado no es posible usar el pulsador P2. Si ocurriera una parada de emergencia debe retomarse el llenado tras restaurar el sistema.

- Cuando el usuario pulse P2 la bomba B2 debe activarse siempre y cuando en el tanque existan más de 10L y se detecte un recipiente en la zona de descarga. La bomba B2 debe apagarse de forma automática tras haber descargado 10L. Si ocurre una parada de emergencia el proceso debe retomarse y continuar la descarga tras restaurar el sistema.
- Los indicadores LED muestran si el sistema se encuentra funcionando, en estado de parada, llenando o descargando.

4.2.1. Program Tags

En los Program Tags se declara todas las variables que se emplearán en la resolución de este Ejemplo. Para esto se crea la Tabla 39 en donde se especifican los Tags y direcciones respectivas de las variables principales.

Tabla 39: Tags del ejemplo

Tags	Variable	Dirección
Bomba1	Bool	DO:2/0
Bomba2	Bool	DO:2/1
S_ULT	Int	AI:3/0
STOP	Bool	DI:1/0
FILL	Bool	DI:1/1
S_CAP	Bool	DI:1/2
S_REC	Bool	DI:1/3
L_STP	Bool	DO:2/2
L_RUN	Bool	DO:2/3
L_REFILL	Bool	DO:2/4
L_FILL	Bool	DO:2/5

Agregamos los Tags principales en AuroraLD Studio en la tabla de datos del programa tal como se muestra en la Figura 341.

Program Tags

N	TAG	VARIABLE	ADDRESS	VALUE	DESCRIPTION
0	Bomba1	BOOL	DO:2/0	0	Bomba 1 (B1)
1	Bomba2	BOOL	DO:2/1	0	Bomba 2 (B2)
2	S_ULT	INT	AI:3/0	0	Sensor Ultrasonico (NIVEL)
3	STOP	BOOL	DI:1/0	0	Pulsador de parada emergencia
4	FILL	BOOL	DI:1/1	0	Pulsador para llenar recipientes
5	S_CAP	BOOL	DI:1/2	0	Sensor Capacitivo (10L)
6	S_REC	BOOL	DI:1/3	0	Sensor para detectar recipiente
7	L_STP	BOOL	DO:2/2	0	Indicador LED - Sistema detenido
8	L_RUN	BOOL	DO:2/3	0	Indicador LED - Sistema funcionando
9	L_REFILL	BOOL	DO:2/4	0	Indicador LED - Llenando recipiente 10L
10	L_FILL	BOOL	DO:2/5	0	Indicador LED - Llenando tanque 60L

Figura 332: Tags del ejemplo creados en AuroraLD Studio

4.2.2. Entorno Ladder

En el entorno Ladder se procede a crear la lógica necesaria para que cumpla con las condiciones de funcionamiento empleando para esto distintas instrucciones. El ejemplo no lo detalla, pero vamos a suponer que el sensor ultrasónico marca un valor de entrada de 1023 cuando el tanque se encuentra a capacidad completa y 0 de valor de entrada si el tanque está vacío.

En las primeras líneas del diagrama Ladder vamos a obtener el valor de entrada del sensor ultrasónico y vamos a escalarlo en unidades de volumen y nivel de líquido en el tanque. El proceso de escalamiento se realiza a través de la instrucción SCP. Lo detallado anteriormente se muestra en la Figura 333.

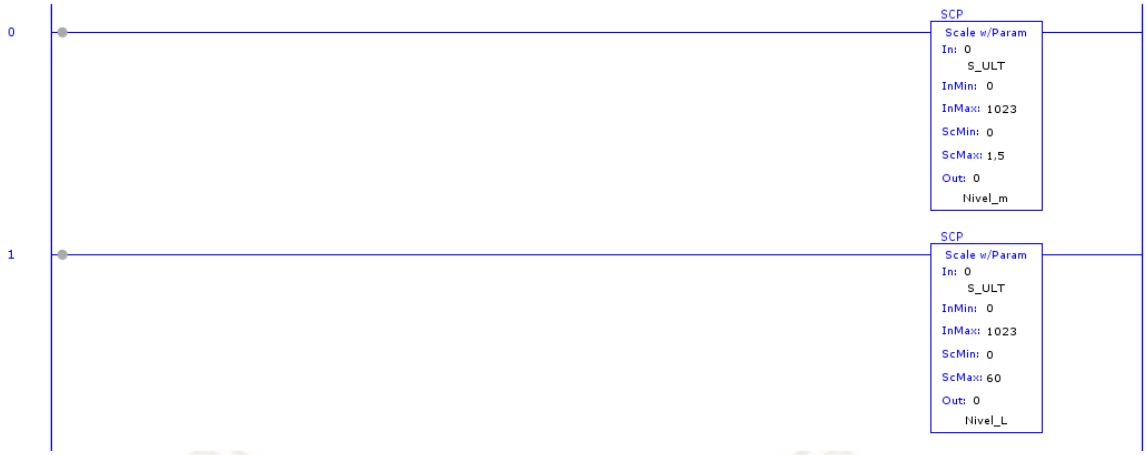


Figura 333: Valor escalado del sensor ultrasónico

El resto de la lógica Ladder se implementa en AuroraLD tal y como muestran las siguientes figuras:

Figura 334: Diagrama Ladder Líneas 2-6

- En la línea 2 se coloca un contador de manera de que cuando se pulsa una vez Stop detiene la ejecución de las demás líneas y cuando se pulsa por segunda vez reanuda todo el sistema. El Preset del contador se coloca en 2.
- La línea 3 permite el reseteo del acumulador una vez que se alcanza el valor de Preset.

- La línea 4 enciende el Led indicador Detenido siempre y cuando se cumpla la condición de entrada. La condición de entrada es una instrucción EQU y se activa si el acumulador del contador es 1.
- La línea 5 activa el Led indicador Corriendo (sistema en ejecución) siempre y cuando el led indicador no esté activo.
- La línea 6 activa la variable interna Llenar siempre y cuando la instrucción LEQ se active si el nivel de líquido en el tanque cae por debajo de 10L.

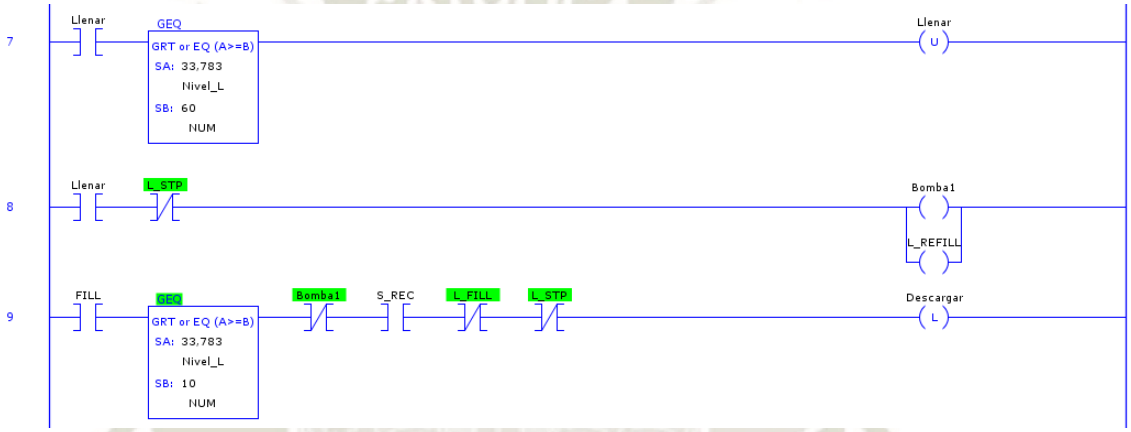


Figura 335: Diagrama Ladder Líneas 7-9

- La línea 7 desactiva la variable interna Llenar una vez que el tanque alcanza un valor de 60L.
- La línea 8 activa la bomba 1 y al mismo tiempo el led indicador de Llenando Tanque.
- La línea 9 permite activar la función de llenar recipiente comprobando la existencia de líquido en el tanque (si es mayor a 10L), la Bomba 1 no está encendida y si existe un recipiente en la zona de descarga.

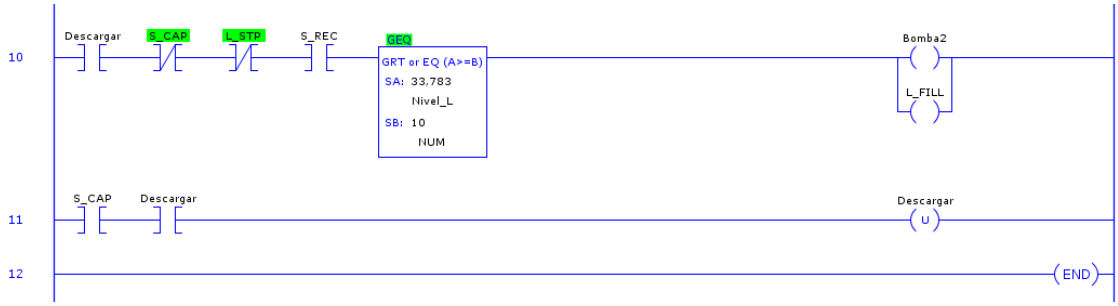


Figura 336: Diagrama Ladder Líneas 10-12

- La línea 10 activa la Bomba 2 y el Led indicador Llenando Recipiente.
- Finalmente, la línea 11 desactiva la Bomba 2 si el sensor capacitivo detecta la presencia de líquido en el recipiente el cual se ha colocado en la posición en la que detecta 10L.

Tras la implementación del diagrama Ladder se han aumentado algunas variables para tener mayor control del programa por lo que los Tags quedarían como se muestra en la Figura 346.

Program Tags

N	TAG	VARIABLE	ADDRESS	VALUE	DESCRIPTION
0	Bomba1	BOOL	O0:2/0	0	Bomba 1 (B1)
1	Bomba2	BOOL	O0:2/1	0	Bomba 2 (B2)
2	S_ULT	INT	AI:3/0	575	Sensor Ultrasonico (NIVEL)
3	STOP	BOOL	DI:1/0	0	Pulsador de parada emergencia
4	FILL	BOOL	DI:1/1	0	Pulsador para llenar recipientes
5	S_CAP	BOOL	DI:1/2	0	Sensor Capacitivo (10L)
6	S_REC	BOOL	INTERNAL	0	Sensor para detectar recipiente
7	L_STP	BOOL	O0:2/2	0	Indicador LED - Sistema detenido
8	L_RUN	BOOL	O0:2/3	1	Indicador LED - Sistema funcionando
9	L_REFILL	BOOL	O0:2/4	0	Indicador LED - Llenando recipiente 10L
10	L_FILL	BOOL	O0:2/5	0	Indicador LED - Llenando tanque 60L
11	Nivel_m	REAL	INTERNAL	0.8431085	Nivel del tanque en metros
12	Nivel_L	REAL	INTERNAL	33.72434	Nivel del tanque en litros
13	CT_STP	COUNTER	INTERNAL	0	
14	Llenar	BOOL	INTERNAL	0	
15	Descargar	BOOL	INTERNAL	0	
16	Bottle	INT	AI:3/1	418	

Figura 337: Tags finales del ejemplo

4.2.3. Entorno Trends

En el ejemplo podemos monitorear el nivel de líquido en el tanque a través de un gráfico de tendencias. Primero lo haremos directamente en AuroraLD Studio y luego lo exportaremos a Excel para verificar la data obtenida.

Se va a monitorear el nivel de líquido en el tanque expresado en litros y se ha configurado la variable número 4 del entorno de tendencias para mostrar el cambio de nivel en el tanque.

La visualización del entorno de tendencias se cambia a automático y procedemos a hacer click en la opción Plot obteniendo el resultado mostrado en la Figura 347.

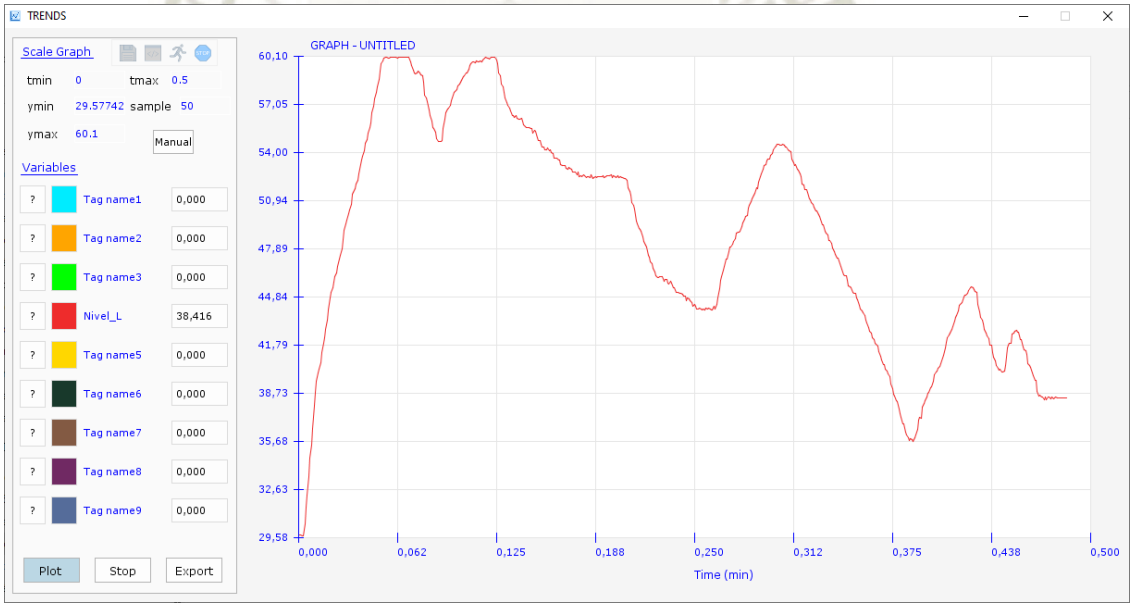


Figura 338: Nivel de líquido en el tanque

Se ha variado el nivel de líquido en el tanque para tener una gráfica con varios cambios. Con los datos obtenidos procedemos a exportar la data a Excel asignando un nombre al archivo:

Figura 339: Exportar datos a Excel

El archivo exportado es un archivo con extensión .csv el cual puede ser importado en Microsoft Excel. Los datos exportados incluyen las 9 variables que se pueden monitorear en el entorno de tendencias y un tiempo expresado en milisegundos en función del tiempo de muestreo configurado. Parte de los datos exportados a Excel se muestran en la Figura 349.

Time (ms)	Nivel_L
0	29.73607
50	29.73607
100	29.73607
150	29.67742
200	29.67742
250	30.439882
300	31.495602
350	32.258064
400	33.431084
450	34.604107

Figura 340: Data exportada a Excel

Con estos datos se puede hacer un gráfico de dispersión como el mostrado en la Figura 350 y obtener un resultado similar al observado en el entorno de tendencias de AuroraLD Studio.

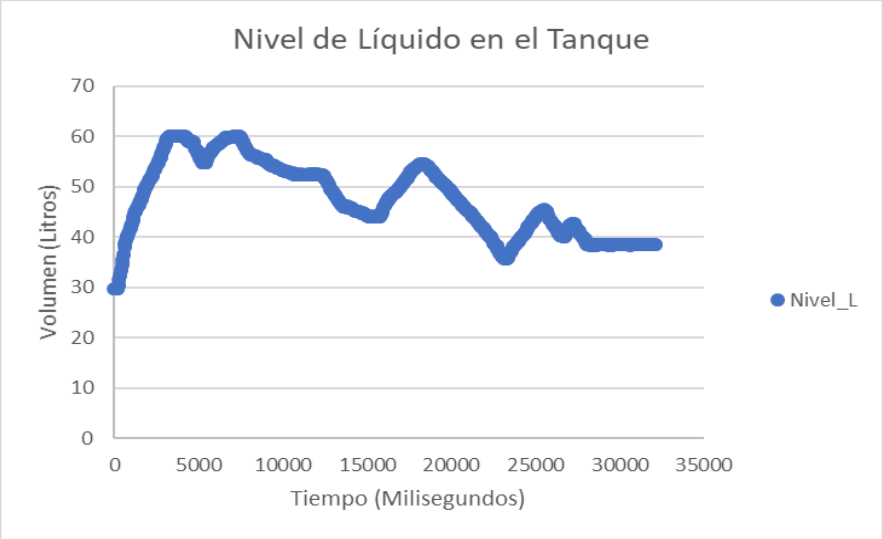


Figura 341: Gráfico de dispersión a partir de los datos exportados

4.2.4. Entorno SCADA

Empleando el entorno de trabajo SCADA podemos crear un sistema de supervisión y adquisición de datos para el proceso del ejemplo. En dicho entorno se usarán los elementos predefinidos en AuroraLD Studio logrando implementar el SCADA mostrado en la Figura 351.

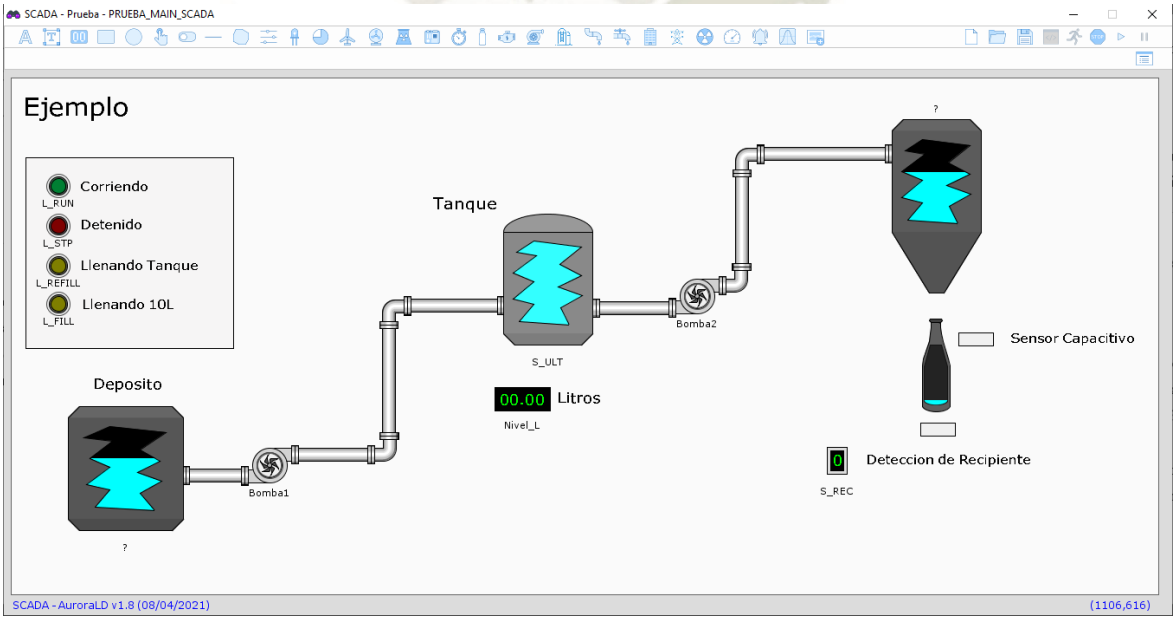


Figura 342: Sistema SCADA modo edición

El sistema SCADA mostrado en la figura cuenta con 3 tanques de los cuales el tanque central es el principal. Se tiene también una caja de texto con una variable de salida asociada de manera que muestre en todo instante de tiempo la cantidad de liquido expresada en litros.

En la parte izquierda se tiene el panel con los indicadores Leds y en el lado derecho se tiene el sistema de llenado. El recipiente de 10L está representando por una botella. Adicionalmente se han agregado rectángulos para que representen a los sensores capacitivos y sensor de presencia de botella.

Se ha empleado la animación visibilidad en la botella vinculado a la variable de sensor de presencia. De esta manera nos aseguramos de que la botella solo se muestra si el sensor se encuentra activo. En la Figura 343 se muestra el sistema SCADA cuando el proceso se encuentra en ejecución.

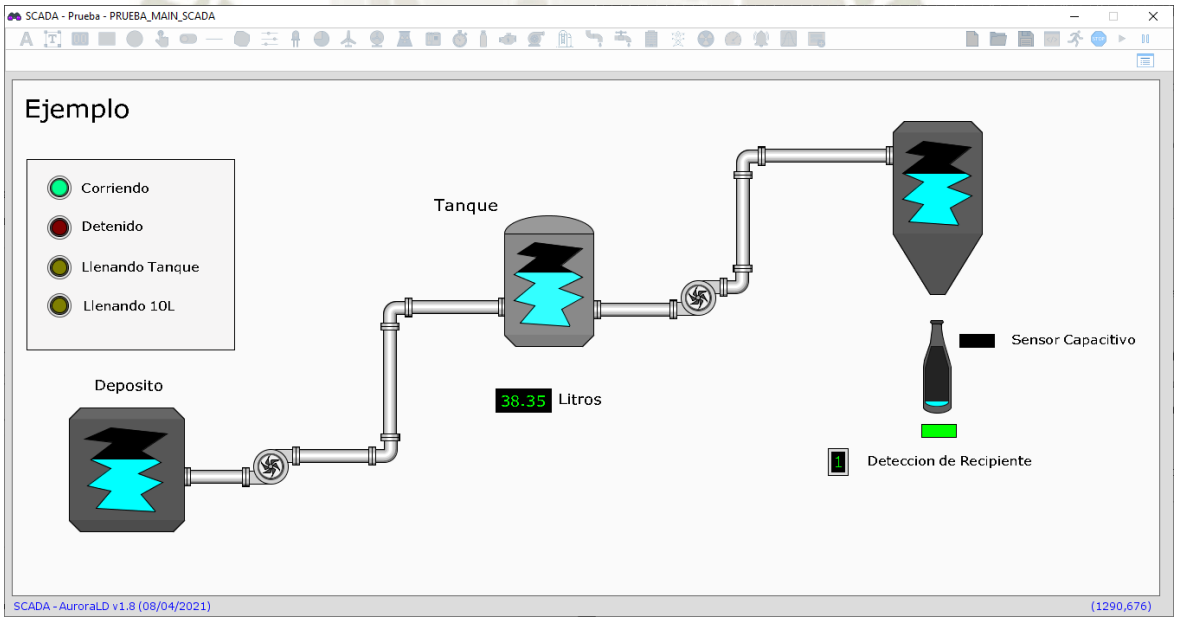


Figura 343: Sistema SCADA en Ejecución

En la Figura 344 se muestra el escenario cuando el tanque tiene un nivel por debajo de 10L.

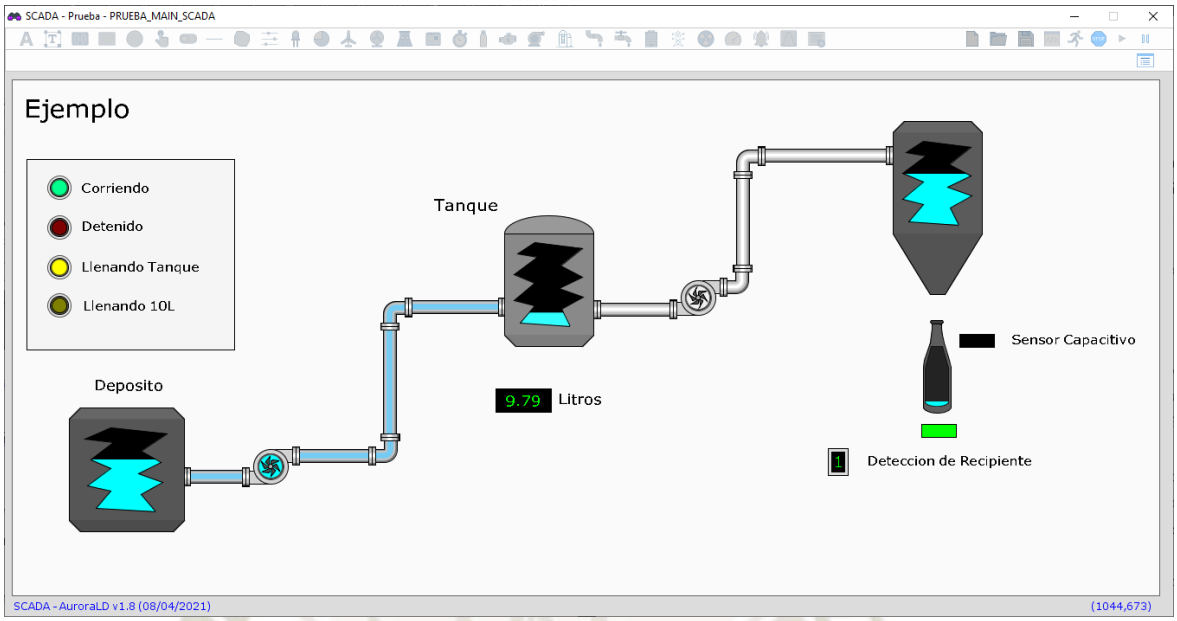


Figura 344: Sistema SCADA llenado de tanque

Bajo estas condiciones notamos que la bomba empieza a trabajar al girar sus álabes en el sistema SCADA, así como también las tuberías cambian de color simulando el líquido que pasa a través de estas con el propósito de llenar el tanque.

Se presenta también en la Figura 345 el escenario en el que se pide al sistema el llenado del recipiente de 10L.

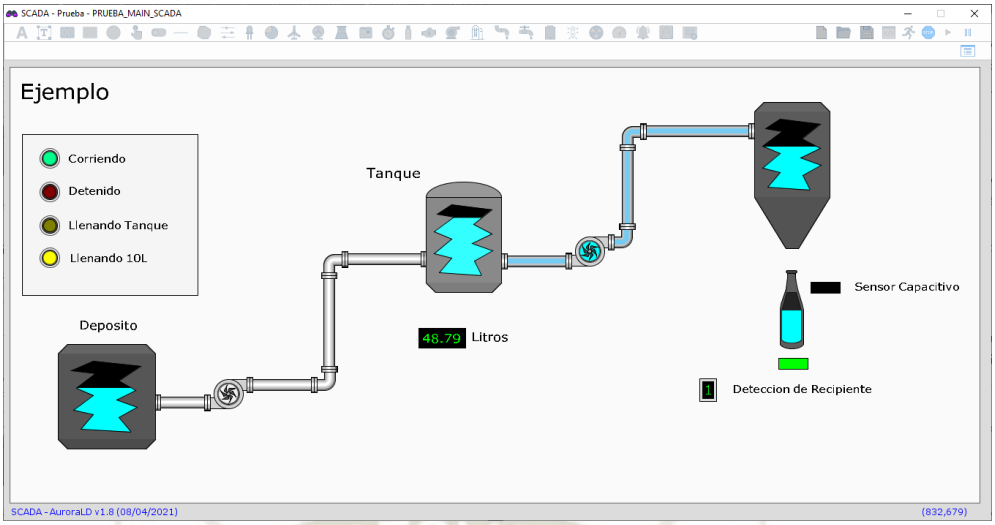


Figura 345: Sistema SCADA llenado de recipiente

En este escenario se presenta la bomba 2 funcionando y las tuberías asociadas a esta bomba se presentan con un cambio de color simulando el líquido que pasa a través de estas. A su vez el recipiente también empieza a llenarse hasta alcanzar y ser detectado por el sensor capacitivo.

Finalmente, también es posible visualizar los cambios de nivel en el tanque a través de gráficos en el entorno SCADA para esto se agrega el elemento Trends y se obtiene el resultado mostrado en la Figura 346.

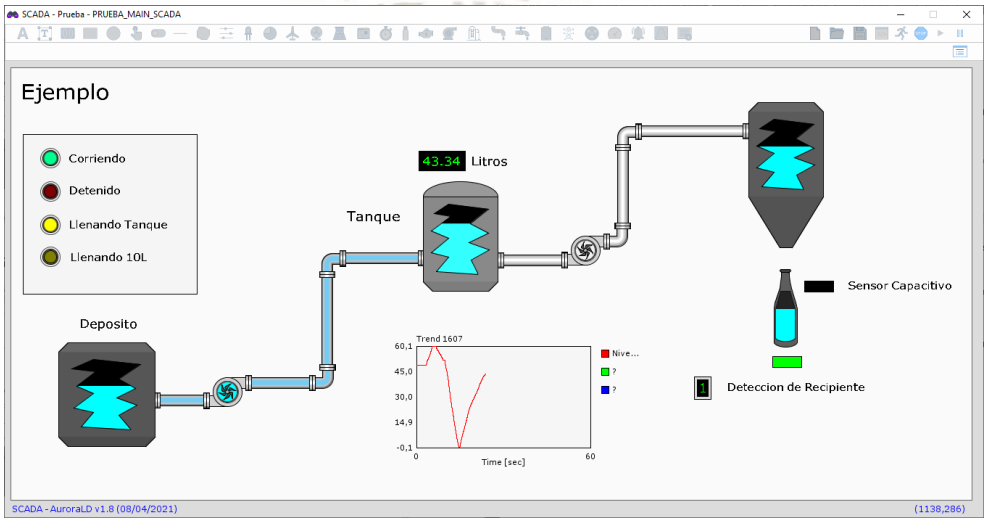


Figura 346: Trends en el sistema SCADA

4.2.5. Circuito Implementado

Para simular el ejemplo planteado se ha conectado dos potenciómetros al PLC1805 v1.0. Estos potenciómetros actúan como los sensores de nivel tanto en el tanque como en el recipiente. Estas entradas son del tipo analógico y por ello sus direcciones están comprendidas por las siglas AI (Analog Input).

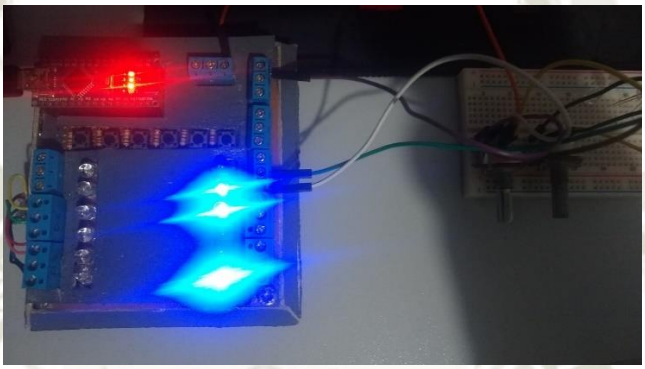


Figura 347: Circuito de ejemplo empleando el PLC1805 v1.0

Los pulsadores incorporados en el PLC1805 v1.0 permiten simular la entrada de los sensores planteados en el ejemplo. Por último, las salidas digitales están representadas por los Leds internos asociados a cada una de las salidas del PLC.

En la Figura 348 se muestra la conexión del ejemplo empleando el PLC1805 v2.0. El desarrollo de una placa facilita a su vez el desarrollo de proyectos con el software. El PLC mostrado en la figura posee las mismas direcciones para las salidas y entradas que la versión 1. También posee la misma cantidad de entradas y salidas con la única diferencia que las entradas son a 12V y en el caso de las salidas: 4 son salidas a transistor de 12V y 2 son salidas a relé que incluso puede manejar cargas en corriente alterna.

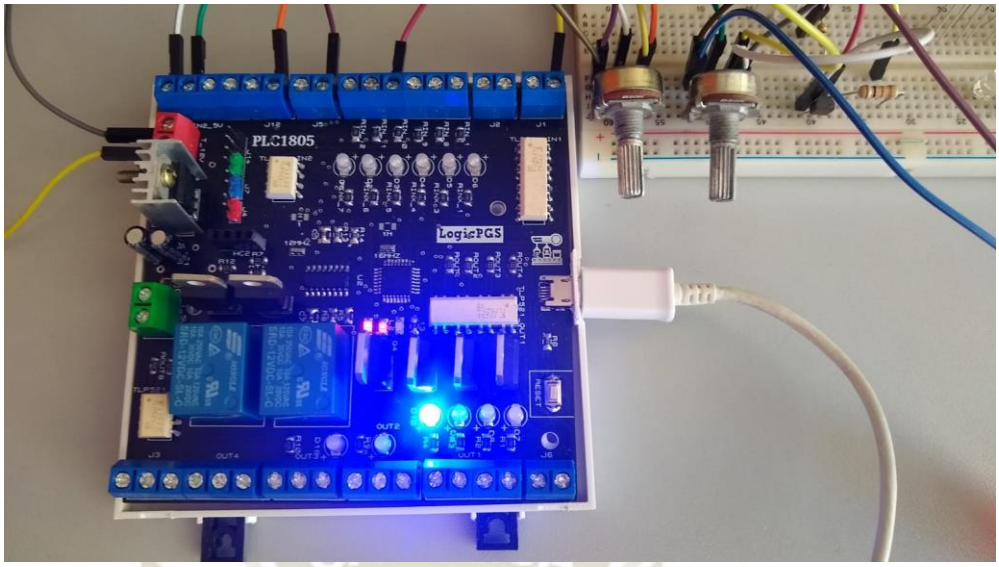


Figura 348: Circuito de ejemplo empleando el PLC1805 v2.0

4.2.6. Ejemplos de otros sistemas SCADA

En las siguientes imágenes se presentan algunos ejemplos respecto a diferentes sistemas SCADA que se pueden implementar en AuroraLD Studio.

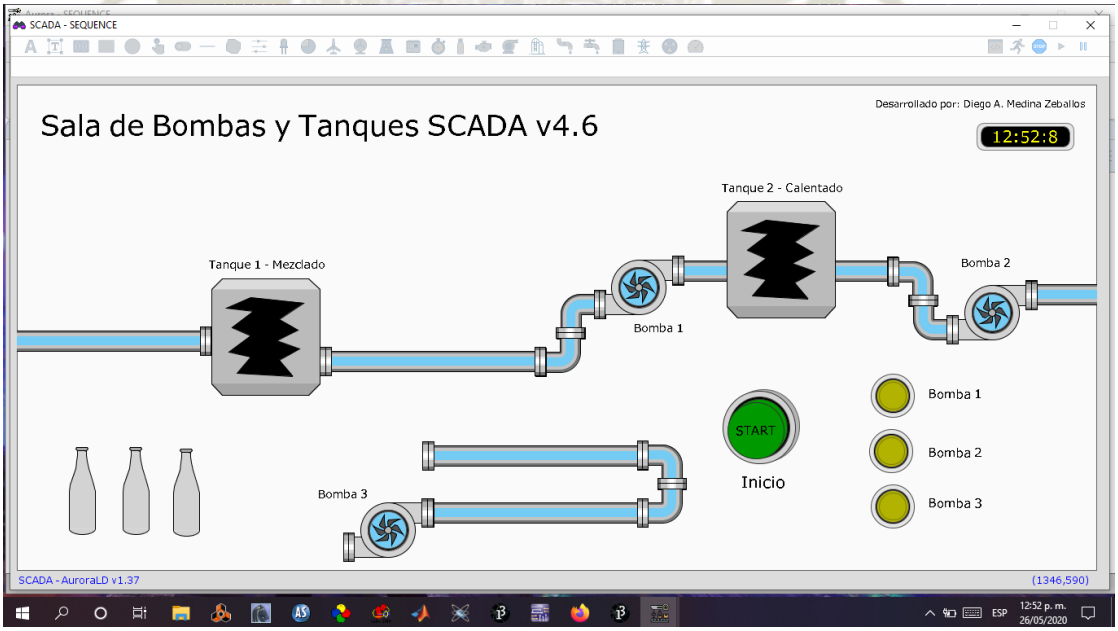


Figura 349: Sistema SCADA Sala de bombas y tanques

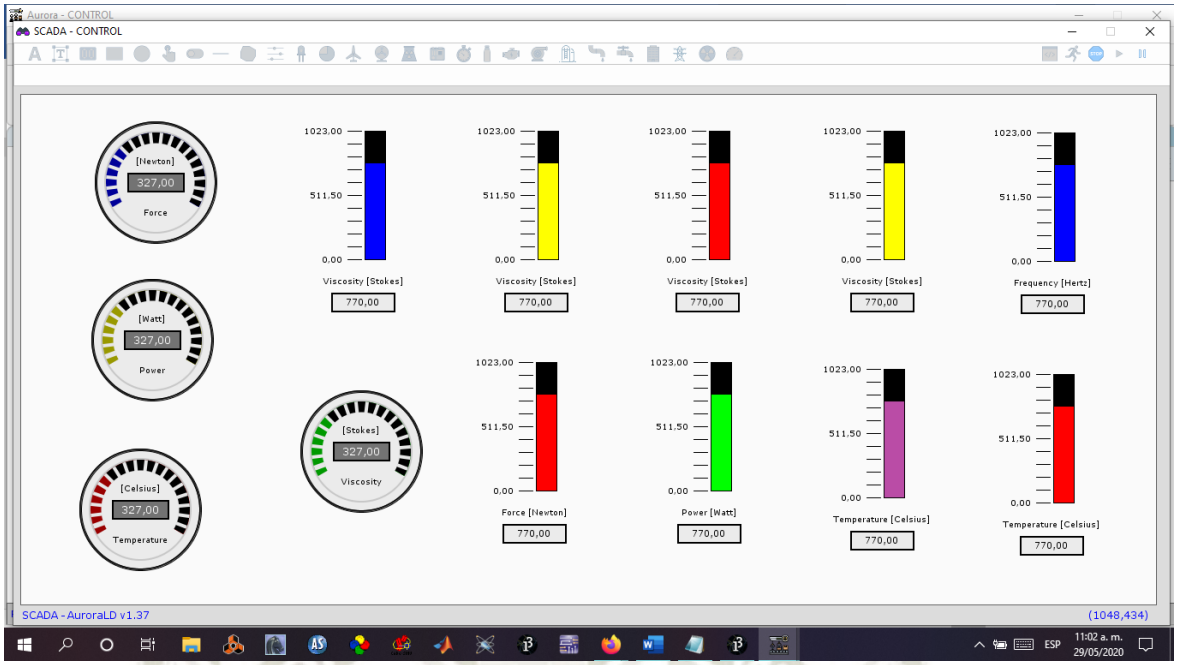


Figura 350: Sistema SCADA panel de variables

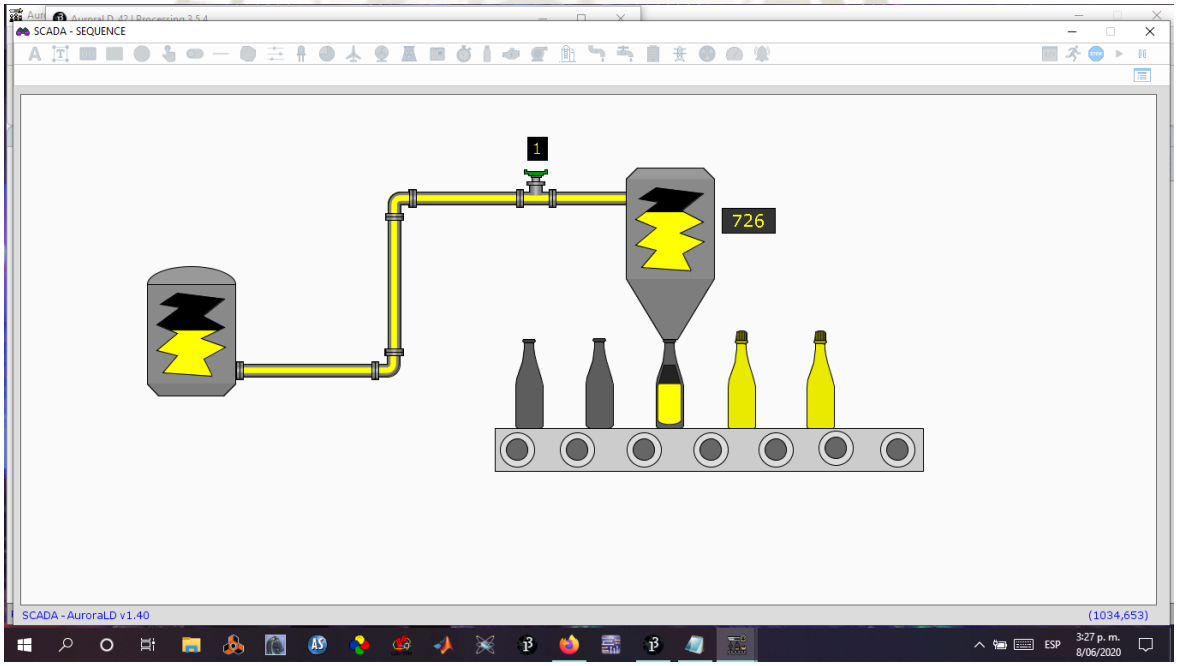


Figura 351: Sistema SCADA llenado de botellas

4.3. Resultados

Con el ejemplo planteado en la parte de pruebas se han empleado distintas herramientas para el desarrollo del ejemplo demostrando así el funcionamiento del software. AuroraLD Studio cumple además con parte del estándar IEC 61131-3.

El entorno Ladder cuenta con instrucciones completamente funcionales que brindan la posibilidad al usuario de implementar cualquier tipo de lógica Ladder empleando dichas instrucciones.

El entorno de tendencias permite monitorear hasta 9 variables distintas ya sean analógicas o digitales permitiendo al usuario incluso exportar los datos monitoreados a Excel para evaluarlo de la forma que se crea conveniente.

Finalmente, el entorno SCADA cuenta con elementos predefinidos que ayudan al usuario a desarrollar un sistema de supervisión y adquisición de datos que faciliten el entendimiento de un proceso.

Resulta complicado realizar distintas pruebas al software ya que las funciones se adaptan a la mayoría de problemas a ser resueltos con lógica Ladder por lo que probar el entorno con un ejemplo se consideró la opción más ideal.

En conclusión, el software implementado cuenta con las herramientas necesarias de manera que ayuden a los estudiantes a desarrollar proyectos de bajo costo empleando placas de desarrollo como Arduino.

CONCLUSIONES

- El software implementado cumple con parte del estándar IEC 61131-3 de acuerdo a lo especificado
- El desarrollo del software está basado en la metodología de desarrollo de software iterativo e incremental, la cual permite probar desde el primer día de desarrollo las funciones que se van agregando al software reduciendo así los errores que puedan surgir y corregirlos en el momento que se presenten.
- Se empleó la programación orientada a objetos para cada una de las instrucciones y elementos con los que cuenta el software facilitando así el manejo de variables.
- En algunas ocasiones el software implementado tiene un consumo elevado de CPU debido a que los distintos elementos, animaciones, ventanas, gráficos, etc. se ejecutan todo el tiempo aun cuando no están siendo utilizados.
- Se redujo el consumo de CPU optimizando algunas partes del código y reduciendo la tasa de refresco de 60 Hz a 30 Hz.
- AuroraLD Studio permite crear un entorno de monitoreo en poco tiempo y con mucha facilidad.
- Se implementó en el software 40 instrucciones Ladder distintas, un entorno de tendencias que permite monitorear hasta 9 variables, así como también un entorno SCADA con 30 elementos diferentes que permiten el desarrollo de una interfaz gráfica para el monitoreo y/o supervisión de procesos.
- Es posible emplear bluetooth para realizar la comunicación con el software, pero presenta fallos al desconectarse y perder la comunicación con el software.
- AuroraLD Studio cuenta con la documentación necesaria para ayudar a entender el manejo del software (videotutoriales y manuales).

RECOMENDACIONES

- Al escribir un software con una interfaz gráfica de usuario es recomendable no ejecutar todas las funciones del programa sino están siendo utilizadas y actualizar cada vez que el usuario realice alguna acción dentro del software reduciendo así el consumo del CPU.
- Resulta importante desarrollar un software que sea completamente independiente a la resolución de la pantalla. Esto permite ejecutar un software en computadoras con baja resolución e incluso asegura el funcionamiento del software con resoluciones actuales que se encuentran en el mercado.
- El desarrollo de proyectos se facilita si es que se crea una placa similar a un PLC para poder hacer pruebas.
- Para solucionar el problema de los tiempos no precisos en los temporizadores se puede optar por implementar un RTC y agregar el código adicional en el software para trabajar con ese dispositivo.
- Implementar un servidor web local que permita monitorear los estados de las variables, así como también el sistema SCADA implementado de forma que se pueda monitorear un proceso de manera local vía web.
- Al establecer una comunicación serial entre el hardware y el software es posible emplear otras placas de desarrollo cambiando variables y modificando parte del código de forma que permitan la comunicación con cualquier placa que posea comunicación serial.

REFERENCIAS

[1] I. Sosa Yarlequé, “Diseño de un programador lógico programable usando microcontrolador atmega y lenguaje Ladder para aplicaciones de laboratorio,” Universidad Nacional de Piura, 2018.

[2] I. V. H. Ivan, P. A. Beatrice, H. R. S. Ivan, and L. C. J. Alejandro, “Design and implementation of a development environment on ladder diagram (HT-PLC) for Arduino with Ethernet connection,” *IEEE ICA-ACCA 2018 - IEEE Int. Conf. Autom. Congr. Chil. Assoc. Autom. Control Towar. an Ind. 4.0 - Proc.*, pp. 1–6, 2019, doi: 10.1109/ICA-ACCA.2018.8609850.

[3] T. Hirata and Y. Hidaka, “Development of Ladder Diagram Software for Learning Programmable Logic Controllers with Arduino,” *J. Polytech. Sci.*, vol. 34, pp. 149–155, 2018, [Online]. Available: http://www.uitec.jeed.or.jp/kenkyu/paper/skill/ka7cok00000004ec-att/Vol.34.S.P149-P155.pdf.

[4] A. Susanto, “Modul Programmable Logic Controller (Plc) Berbasis Arduino Severino,” *Modul Program. Log. Controll. Berbas. Arduino Sev.*, vol. 1, no. 2, 2017, doi: 10.21831/jee.v1i2.17413.

[5] Q. Hidayati, F. Z. Rachman, N. Yanti, N. Jamal, and S. Suhaedi, “Desain Model dan Simulasi PLC-Mikrokontroler sebagai Modul Pembelajaran Berbasis PLC,” *J. Teknol. Rekayasa*, vol. 2, no. 2, p. 73, 2017, doi: 10.31544/jtera.v2.i2.2017.73-82.

[6] I. Nehrutsa and J. Wuesthues, “LDmicro.” 2005, [Online]. Available: https://osimplc.com/docs/es/07-ldmicro.html.

[7] Waltech System, “LadderMaker.” 2015, [Online]. Available: https://waltech.com/cszcms/laddermaker.

[8] Outseal, “Outseal Studio.” [Online]. Available: http://www.outseal.com/.

[9] R. Kazala and P. Straczynski, “The Most Important Open Technologies for Design of Cost Efficient Automation Systems,” *IFAC-PapersOnLine*, vol. 52, no. 25, pp. 391–396, 2019, doi: 10.1016/j.ifacol.2019.12.567.

[10] A. Powell-Morse, “Iterative Model: What Is It And When Should You Use It?,” *Airbrake*, 2016. https://airbrake.io/blog/sdlc/iterative-model.

[11] K.-H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*, 2nd ed. Springer-Verlag Berlin Heidelberg, 2010.

[12] “Processing.org.” https://processing.org/.

[13] D. Shiffman, *Learning Processing: A Beginner’s Guide*. 2008.

[14] D. Poo, D. Kiong, and S. Ashok, *Object-Oriented Programming and Java*, 2nd ed. London: Springer, London, 2008.

[15] “What is object-oriented programming (OOP)?” https://searchapparchitecture.techtarget.com/definition/object-oriented-programming-OOP.

[16] O. de Barraza, F. de Krol, L. de Meléndez, and M. de Velásquez, *Introducción a la Programación Orientada a Objetos*. 2006.

[17] “Ladder Logic Basics - Ladder Logic World.” https://ladderlogicworld.com/ladder-logic-basics/.

[18] “What is SCADA? Supervisory Control and Data Acquisition.” https://inductiveautomation.com/resources/article/what-is-scada.

[19] D. Alison, “The father of invention: Dick Morley looks back on the 40th anniversary of the PLC - Manufacturing AUTOMATION,” *Manufacturing Automation*, 2009. https://www.automationmag.com/855-the-father-of-invention-dick-morley-looks-back-on-the-40th-anniversary-of-the-plc/.

[20] M. G. Hudedmani, R. M. Umayal, S. K. Kabberalli, and R. Hittalamani, “Programmable

Logic Controller (PLC) in Automation," *Adv. J. Grad. Res.*, vol. 2, no. 1, pp. 37–45, 2017, doi: 10.21467/ajgr.2.1.37-45.

[21] "What is Arduino? | Arduino," Feb. 2018. https://www.arduino.cc/en/Guide/Introduction.

[22] J. Mallari, "How to Set Up UART Communication on the Arduino." https://www.circuitbasics.com/how-to-set-up-uart-communication-for-arduino/.

[23] R. Automation and Allen-Bradley, "Logix 5000 Controllers IEC 61131-3 Compliance." p. 32, 2018, [Online]. Available: https://literature.rockwellautomation.com/idc/groups/literature/documents/pm/1756-pm018_-en-p.pdf.

ANEXOS

Anexo A: Software AuroraLD Studio

El software se ha subido a una carpeta de Google Drive. Al acceder a esta carpeta se puede descargar el software para Windows y para Linux. El link respectivo para la descarga del software es el que se muestra a continuación:

https://drive.google.com/drive/folders/10a1SUZ8uz0KZkhRZB_eBf09h15Shshjg?usp=sharing

Al acceder al link se tiene acceso a 5 carpetas, las cuales se muestran en la Figura 352. En el caso de Windows se tienen los instaladores respectivos (32 o 64 bits) y en el caso de Linux se manejan versiones portables.

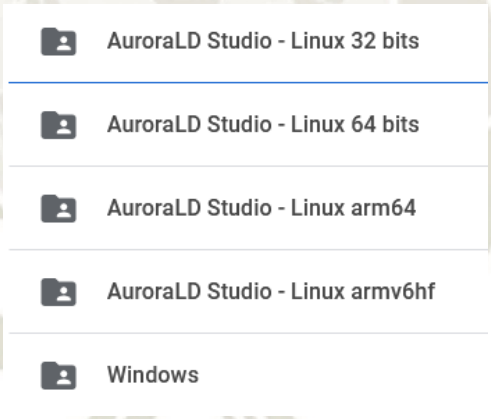


Figura 352: Carpetas del software AuroraLD Studio

El software para Windows ha sido cargado a un repositorio de GitHub en el que se tiene información del software y los instaladores respectivos:

https://github.com/dmedina09/AuroraLD_Studio_Software

Anexo B: Manuales

Se ha desarrollado 4 manuales que ayudan a entender el manejo del software. Estos manuales como se ya especificó con anterioridad están en inglés y también se han cargado a una carpeta en Google Drive tal como se muestra en la Figura 353.

Figura 353: Manuales de AuroraLD Studio

- El manual número uno “UIM001” hace referencia a la instalación del software.
- El manual número dos “UIM002” hace referencia al entorno Ladder.
- El manual número tres “UIM003” hace referencia al entorno de tendencias.
- El manual número cuatro “UIM004” hace referencia al entorno SCADA.

Estos manuales se encuentran en la carpeta de instalación del software, pero también se pueden acceder a ellos a través del siguiente enlace:

https://drive.google.com/drive/folders/12iy2vozK5bXCx3IkX-c35IdCF35Qo8wm?usp=sharing

Anexo C: Videotutoriales

Se ha desarrollado 15 videotutoriales los cuales forman parte de una serie de videos que ayudan a entender las distintas funciones del software. Los videos han sido cargados a un canal de Youtube que lleva por nombre **LogicPGS**. El acceso al canal es público y se puede acceder al canal buscando directamente en Youtube o a través del siguiente enlace:

https://www.youtube.com/channel/UCBwRfFjgCiSxVayGX6VGDiw

Dentro del software en la opción de ayuda también es posible acceder a los videotutoriales y al hacer click se abrirá automáticamente en el navegador el canal en mención.

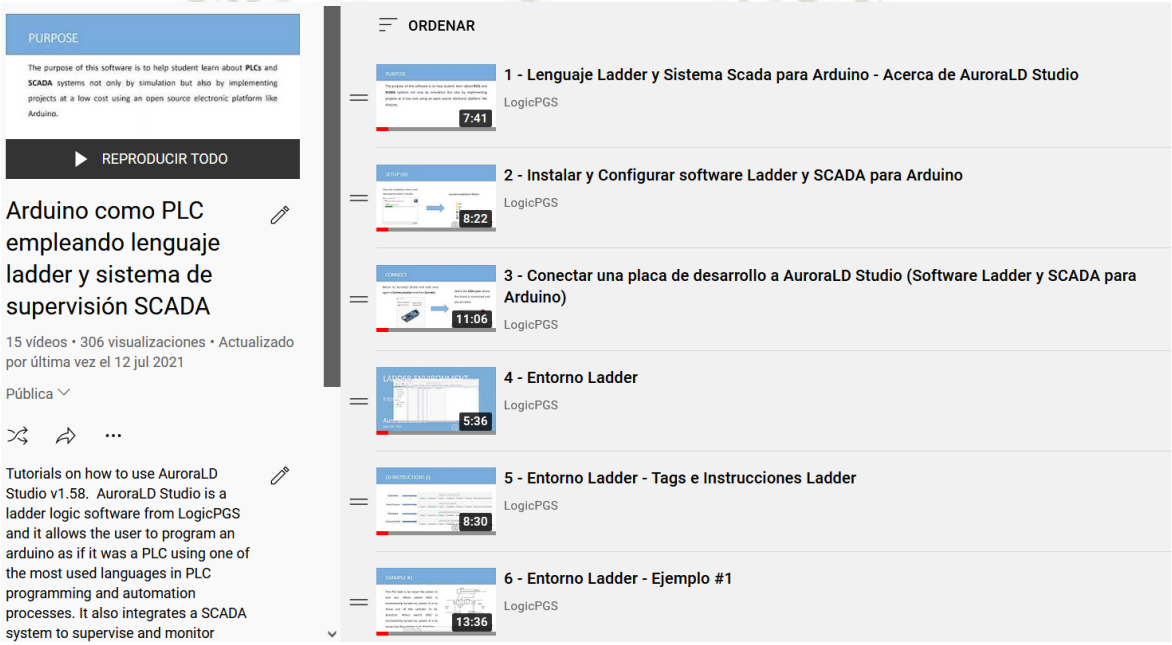
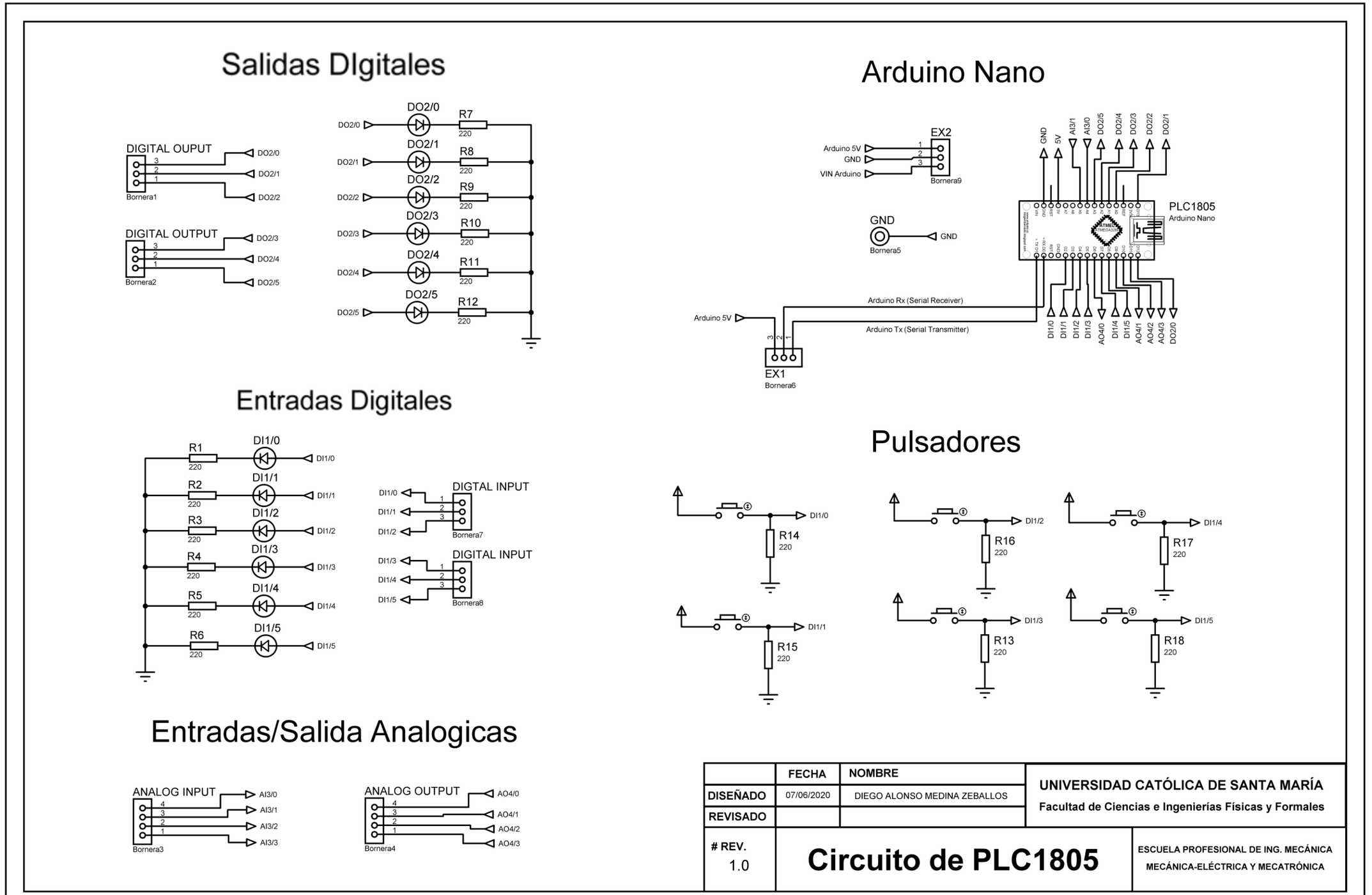


Figura 354: Lista de reproduccion creada en el canal de youtube

En los videos se presenta de fondo slides en inglés sin embargo el audio de todos los videos está completamente en español.

Anexo D: Circuito de PLC1805 v1 (Módulo de pruebas)



	FECHA	NOMBRE	UNIVERSIDAD CATÓLICA DE SANTA MARÍA
DISEÑADO	07/06/2020	DIEGO ALONSO MEDINA ZEBALLOS	Facultad de Ciencias e Ingenierías Físicas y Formales
REVISADO			
# REV. 1.0	**Circuito de PLC1805**		ESCUELA PROFESIONAL DE ING. MECÁNICA MECÁNICA-ELÉCTRICA Y MECATRÓNICA