UNIVERSIDAD DEL AZUAY

FACULTAD DE CIENCIA Y TECNOLOGÍA

ESCUELA DE INGENIERÍA ELECTRÓNICA

Diseño e Implementación de un Sistema SCADA mediante Protocolo ModBus con Comunicación Inalámbrica para el Control de un Robot

Trabajo de graduación previo a la obtención del título de:

INGENIERO ELECTRÓNICO

Autores:

DALTON DEMETRIO TOLEDO TORRES

PABLO DARIO URGILÉS CÁRDENAS

Director:

ING. HUGO MARCELO TORRES SALAMEA Ph.D

CUENCA - ECUADOR

2017

DEDICATORIA:

Dedico este trabajo a mi madre Lourdes, por ser el pilar más importante, por demostrar siempre su cariño y apoyo incondicional.

A mi padre Joselito, por ser el ejemplo de perseverancia, por apoyarme cuando he estado más vulnerable y caído.

A mis hermanos por su apoyo intelectual y sentimental, a mi cuñada y mis hermosas sobrinas por siempre sacarme una sonrisa con su cariño sincero.

Con todo mi cariño a mis abuelitos y a mis tíos por motivarme, aconsejarme y ser una parte fundamental para formarme como una persona de bien.

Darío Urgilés C.

DEDICATORIA:

Este trabajo va dedicado a toda mi familia ya que han sido el apoyo principal durante toda mi carrera universitaria, en especial a mi madre Cecilia y a mi padre Demetrio que son las personas que siempre han estado conmigo en los momentos difíciles y en los momentos de felicidad.

A mis hermanos Carolina, Andy y Sebastián que me han ayudado de diferentes formas para lograr culminar esta etapa de mi vida.

A mi sobrina Danna por darme más felicidad en mi vida.

Dalton Toledo.

AGRADECIMIENTOS:

Queremos expresar nuestros profundos agradecimientos a los profesores de la Escuela de Ingeniería Electrónica, por sus enseñanzas a lo largo de nuestros estudios.

Al Ing. Juan José Vidal por su esfuerzo y dedicación, quien con sus conocimientos y su motivación ha logrado que podamos terminar nuestros estudios con éxito.

Además de manera especial un agradecimiento a nuestro director de tesis, Ing. Hugo Marcelo Torres Salamea Ph.D, por darnos la idea de plasmar el tema logrado, por su rectitud en su profesión como docente, por sus consejos, y por su paciencia para lograr culminar con la tesis planteada.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ANEXOS

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA SCADA MEDIANTE PROTOCOLO MODBUS CON COMUNICACIÓN INALÁMBRICA PARA EL CONTROL DE UN ROBOT

RESUMEN

El presente trabajo de investigación se fundamenta en la implementación de un sistema de control para un robot de 3GDL mediante el protocolo ModBus.

Para la comunicación entre el manipulador y el sistema SCADA se utilizó el código Hamming como técnica de detección y corrección de errores, este se implementó en lenguaje Python.

El sistema de enlace que se utilizó, fue mediante módulos Xbee, que permitieron la comunicación inalámbrica entre un PC que hace la función de codificador y una tarjeta PcDuino utilizado como decodificador. Para el movimiento de los servo motores del robot, se utilizó una tarjeta Expander Pi.

Palabras claves: ModBus, Xbee, SCADA, Hamming, Robot.

Ing. Hugo M. Torres Salamea Ph.D.

Director del Trabajo de Titulación

Ing. Daniel E. Iturralde Piedra Ph.D.

Coordinador de Escuela

Dalton Demetrio Toledo Torres

Autor

Pablo Darío Urgilés Cárdenas

Autor

DESIGN AND IMPLEMENTATION OF A SCADA SYSTEM BY MODBUS PROTOCOL WITH WIRELESS COMMUNICATION FOR THE CONTROL OF A ROBOT

ABSTRACT

This research work was based on the implementation of a control system for a 3 DOF robot using Modbus protocol. The Hamming code, implemented in Python language, was used as a detection and correction technique for the communication between the manipulator and the SCADA system. The link system used was through Xbee modules, which allowed wireless communication between a PC that performs the encoder function, and a pcDuino card used as a decoder. An Expander Pi card was used for the movement of the robot's servo motors.

Keywords: Modbus, Xbee, SCADA, Hamming, Robot.

Ing. Hugo M. Torres Salamea Ph.D.
Thesis Director

Ing. Daniel E. Iturralde Piedra Ph.D.
School Director

Dalton Demetrio Toledo Torres

Pablo Darío Urgilés Cárdenas

Authors

UNIVERSIDAD DEL AZUAY
Idiomas

Translated by,
Lic. Lourdes Crespo

Toledo Torres Dalton Demetrio

Urgilés Cárdenas Pablo Darío

Trabajo de Titulación

Ing. Hugo Marcelo Torres Salamea Ph.D.

Abril, 2017.

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA SCADA MEDIANTE PROTOCOLO MODBUS CON COMUNICACIÓN INALÁMBRICA PARA EL CONTROL DE UN ROBOT

INTRODUCCIÓN

En la industria existe un alto índice de automatización, debido a esto la tecnología avanza exponencialmente, generando que se desarrollen diversos protocolos industriales de comunicación, estos facilitan el control o supervisión remota de un proceso productivo. Uno de los problemas en los sistemas de comunicaciones industriales son los medios de transmisión de datos, por ejemplo los sistemas alámbricos que desperdician recursos físicos al necesitar construir una red que comunique al receptor con el emisor y necesitan de una buena planificación para el tendido de los cables para que estos funcionen correctamente.

El objetivo del presente trabajo es “Diseñar e implementar un sistema SCADA para el control de un robot mediante una comunicación industrial basado en el protocolo ModBus”. Para cumplir con el objetivo se utilizó la siguiente metodología:

Se realizó un estudio de los protocolos de comunicaciones industriales, para lo cual fue necesario investigar las características que presentan cada uno de ellos como también acudir a diferentes publicaciones relacionadas con protocolos industriales utilizados para manipular robots.

Luego se realizó un estudio detallado del protocolo ModBus debido a las características de fácil interacción con el usuario, para lo cual se implementó la

comunicación mediante este protocolo entre maestro y esclavo con la ayuda de simuladores virtuales.

Además de la comunicación mediante el protocolo ModBus, se implementó un sistema tolerante a fallas basado en código Hamming, los mismos que fueron desarrollados en Python con la ayuda de dos tarjetas PcDuino que hacen las veces de codificador y decodificador Hamming.

Dentro de la metodología utilizada, se implementó un sistema SCADA para la comunicación ModBus entre el codificador y decodificador Hamming, para lo cual fue necesario utilizar el software Blender debido a las características que este presenta.

Para verificar el correcto funcionamiento del sistema y dar cumplimiento con el objetivo planteado, fue necesario realizar las pruebas de comportamiento del manipulador de 3GDL, siendo necesario implementar el seguimiento de una trayectoria por parte del robot frente a datos que contienen errores, supervisando continuamente el movimiento del manipulador.

CAPÍTULO 1

PROTOCOLOS DE COMUNICACIÓN INDUSTRIALES Y PRINCIPALES CARACTERÍSTICAS DEL PROTOCOLO MODBUS

1.1 Introducción a los protocolos de comunicación industriales.

Un protocolo de comunicación abarca todas las reglas y convenciones que deben seguir dos equipos para poder intercambiar información.
La estructura de cualquier tipo de enlace de comunicación se puede elaborar como se indica en la figura 1.1:

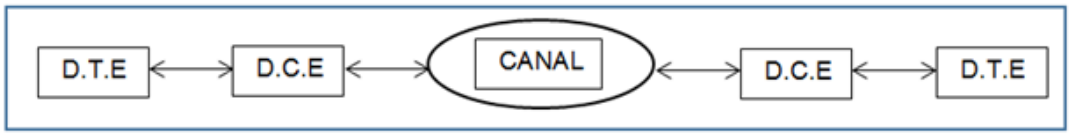


Figura 1.1: Componentes de un enlace de datos.

DTE (*Data Terminal Equipment*): Equipo Terminal de Datos.
DCE (*Data Communication Equipment*): Equipo de Comunicación de Datos.

El objetivo de cualquier protocolo de comunicación es poder conectar y mantener el diálogo entre dos Equipos Terminales de Datos (DTE), permitiendo que la información pueda fluir entre ambos con seguridad (sin fallos), es decir, todas las reglas y especificaciones del lenguaje a utilizar por los equipos.
La estandarización es un punto de conflicto entre intereses técnicos y comerciales, ya que cada fabricante realiza sus investigaciones de acuerdo a las necesidades de sus equipos, después se pretende que estas utilidades se conviertan en estándar (Penin A. R., Sistemas SCADA, 2007).

Este tipo de protocolos tienen denominaciones tales como:

Hart[1]	Control de Procesos
Profibus	Control Discreto y Control de Procesos

[1] Hart: *Highway Addressable Remote Transducer*

AS-i[2]	Control Discreto
Can[3]	Control Discreto

Un protocolo puede integrarse, en mayor o menor medida, en cualquier nivel de la famosa Pirámide de Automatización (CIM[4]), pero el objetivo está en encontrar la relación prestaciones/precio ideal, y el equilibrio entre varias tecnologías que permiten complementarse unas a otras.

Dependiendo de la aplicación existen buses[5] más adecuados que otros, es decir no existe un bus mejor que otro. Para decidir que bus utilizar en cada aplicación, es conveniente tener en cuenta los siguientes puntos:

- Costo de nodo por bus.
- Costo de programación (o desarrollo).
- Tiempos de respuesta.
- Fiabilidad.
- Robustez (tolerancia a fallos).
- Modos de funcionamiento (Maestro-Esclavo, acceso remoto).
- Medios Físicos (cable, fibra óptica, radio, etc.)
- Topologías permitidas.
- Gestión.
- Interfaces de usuario.
- Futuro (o lo que es lo mismo normalización).

1.2 Tipos de redes según su forma (Topología).

La Topología define la disposición de los diferentes equipos alrededor del medio de transmisión de datos, determinando una estructura de red característica (Penin A. R., Sistemas SCADA, 2007).

[2] AS-i: *Actuador Sensor Interface*

[3] Can: *Controller Area Network*

[4] CIM: *Computer Integrated Manufacturing*

[5] Bus: Bus de datos o canal es un sistema digital que transfiere datos entre varios componentes electrónicos.

1.2.1 Redes Centralizadas (*Clustered Systems*).

- Todos los equipos dependen de un equipo central (*Host*) el mismo que controla todo el sistema. El *Host* debe ser un equipo potente para poder gestionar el tráfico de datos con eficiencia.
- El fallo de un terminal no afecta al funcionamiento de la red, si hay un fallo en el *host* el sistema se paraliza por completo.

1.2.2 Redes Distribuidas (*Distributed Systems*).

- En este tipo de red, los equipos puede ser máquinas sencillas que comparten las cargas de trabajo, los recursos y las comunicaciones.
- El fallo de un terminal no afecta al resto de equipos.

Las redes centralizadas se basan en la potencia del *Host*, y las redes distribuidas se basan en la distribución de los equipos menos potentes, pero con mucha más versatilidad ya que son más tolerantes a fallos.
Existen varias configuraciones básicas:

- Anillo
- Estrella
- Bus
- Árbol
- Red
- Híbrida

1.2.2.1 Anillo

El medio de transmisión forma un circuito cerrado anillo (ver figura 1.2) al que se conectan los equipos (Shah, 2017).

Las principales características de este tipo de topología son:

- Los requerimientos de cable son mínimos.
- Se basa en una serie de conexiones punto a punto de una estación con la siguiente.
- El modo de transmisión se organiza por turnos mediante el paso de un permiso de transmisión de una estación a otra.
- El mensaje vuelve al emisor.

- El tráfico de la información tiene un sentido único a lo largo del soporte de transmisión.
- La señal se regenera en cada nodo, es una estructura activa.
- No permite la ampliación en funcionamiento (se interrumpiría físicamente la red).

Posibles desventajas:

- La caída de un equipo interrumpe el tráfico de información.
- Diagnóstico difícil debido al sentido único de flujo de información.
- Añadir o quitar nodos afecta la red.
- Distancias limitadas entre nodos.

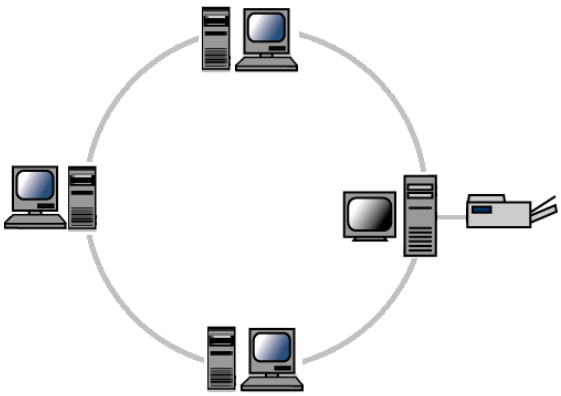


Figura 1.2: Topología de Anillo.
Fuente: (Malavolta, 2007).

1.2.2.2 Estrella

En esta configuración, todos los equipos están conectados a un equipo o nodo central (*HUB Host Unit Broadcast*) que realiza las funciones de control y coordinación (ver figura 1.3) (Hussein, Jakllari, & Paillasa, 2015).

Las principales características son las siguientes:

- La transmisión de información es punto a punto.
- Mantenimiento simple.
- El equipo central (HUB) controla toda la red.

- El rendimiento de la red dependerá del HUB.
- La caída de un equipo no afecta al resto.
- Diagnóstico de fallos sencillo.
- Si el HUB se paraliza la red se detiene totalmente.
- Se necesita más conductores (cables) que en otras topologías.
- La ampliación del sistema está limitada por la capacidad del HUB.

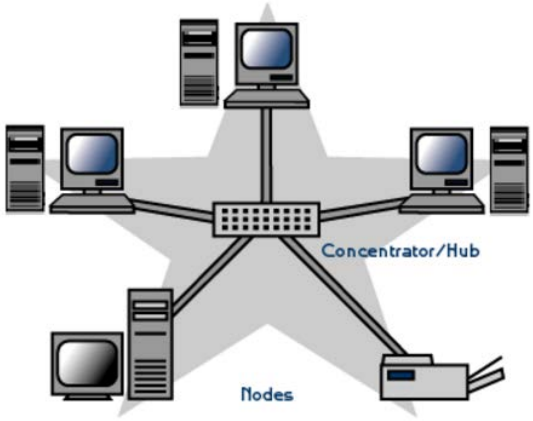


Figura 1.3: Topología Estrella.
Fuente: (Malavolta, 2007).

1.2.2.3 Bus

La distribución básica se realiza alrededor de un segmento de conductor al cual se conectan todos los equipos como se muestra en la figura 1.4.

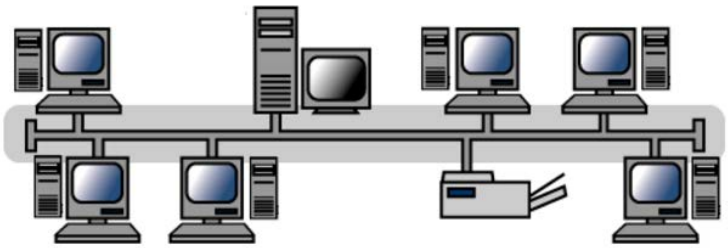


Figura 1.4: Topología tipo Bus.
Fuente: (Malavolta, 2007).

El modo de transmisión es aleatorio, un equipo transmite cuando lo necesita. Si existen colisiones, se cuentan con algoritmos que nos ayudan a resolver este tipo de inconvenientes.

Las características más destacables son las siguientes:

- Necesita menor longitud de conductor en comparación con otros tipos de topología.
- Las conexiones de alta impedancia permiten conectar y desconectar elementos de forma sencilla, la caída de un equipo no afecta al resto de la red.
- Velocidad de transmisión elevada.
- Todos los equipos pueden transmitir a cualquier otro equipo según lo necesiten (conexión multipunto).
- Número reducido de conexiones.
- Ampliación sencilla.
- Es la opción más extendida en buses de campo.

Posibles desventajas:

- Falta de seguridad, cualquier nodo puede ver cualquier mensaje aunque no sea el destinatario.
- Debido a la estructura física el diagnóstico puede ser difícil, un fallo eléctrico puede estar en cualquier punto del bus
- No hay reconocimiento de mensajes automático (no vuelven al emisor).
- En casos de sobrecarga de tráfico puede bajar el rendimiento (Hussein, Jakllari, & Paillasa, 2015).

1.2.2.4 Árbol

Tiene las características de las tres topologías anteriores (ver figura 1.5).

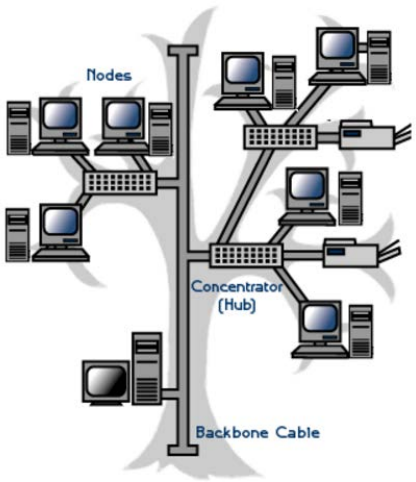


Figura 1.5: Topología tipo Árbol.
Fuente: (Malavolta, 2007).

Se encuentra en los sistemas de bus tipo sensor-actuador (AS-i).

1.2.2.5 Red

Permite la conexión entre dos estaciones a través de múltiples caminos.
Características principales:

- Fiabilidad y tolerancia a fallos. Si una línea de transmisión se daña, el tráfico es redirigido por otro camino sin afectar la comunicación.
- Alto costo de implementación.
- No utilizado en buses de campo.

A nivel industrial las topologías más usadas son las de Bus y de Anillo, debido a su robustez ante fallos, velocidad de transmisión y sencillez de ampliación (Penin A. R., 2012).

1.2.2.6 Híbrida

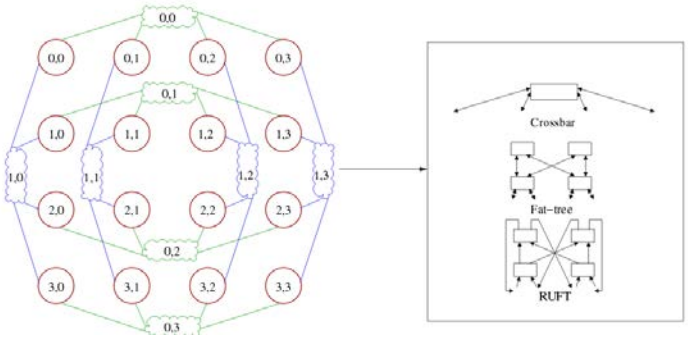


Figura 1.6: Topología tipo híbrida.
Fuente: (Cebrián, 2012).

Es una red que usa una o más topologías de manera que no tiene algún estándar especifico, se puede nombrar topología hibrida a la conexión de dos tipos de redes mencionadas como el caso del tipo estrella y árbol (ver figura 1.6).

1.3 Sistemas de control industrial

Podemos nombrar tres tipos de sistemas de control industrial: control centralizado, control híbrido y control distribuido. Para determinar qué tipo de control se debe aplicar a un sistema en concreto debemos tener en cuenta algunas de las características del sistema, las cuales pueden ser: la importancia de las tareas a realizar, la posibilidad

de subdividir la tarea de control del proceso o conjunto de máquinas en funciones autónomas (Muñoz, 2009).

1.3.1 Control Centralizado

Este tipo de control es el caso de sistemas poco complejos donde los procesos pueden ser gestionados mediante un único elemento de control encargado de realizar todas las tareas del proceso de producción y que puede incluir un sistema de monitorización y supervisión. A medida de que las necesidades de producción requieren mayor complejidad, la tendencia ha sido emplear elementos de control más complejos y potentes, manteniendo en único elemento de control en todo el proceso.

La ventaja de esta metodología es principalmente que no se necesita planificar un sistema de intercomunicación entre procesos, ya que todas las señales estas gestionadas por el mismo sistema. Así mismo, el sistema tiene varias desventajas, ya que si el sistema principal falla, se paralizan todas las comunicaciones, esto se podría resolver con un sistema redundante. También es necesario el empleo de unidades de control (generalmente autómatas programables) de mayor capacidad de proceso dada la complejidad de los problemas que debe abordar y con las restricciones de tiempo límite que son habituales en los procesos industriales debido a la sincronización necesaria; pueden existir problemas de tiempos de ciclo en el caso de procesos muy complejos.

Por último el cableado aumenta notablemente debido a las distancias que existen entre los sensores, actuadores y la unidad de control, este problema se puede simplificar de cierta forma con el uso de buses de campo (polverini, 2017).

1.3.2 Control Distribuido

El control distribuido requiere que puedan considerarse procesos, grupos de procesos o áreas funcionales susceptibles de ser definidas por un algoritmo de control que pueda realizarse de forma autómata. Cada unidad contará con un elemento de control (o autómata) de acuerdo con los requerimientos del proceso. Debido a la interdependencia que existe entre las operaciones, es necesario interconectar los elementos de control entre sí mediante entradas y salidas digitales o, a través de una red de comunicaciones para intercambio de datos y estados, por lo tanto el elemento de control debe permitir las comunicaciones.

Con esta metodología de control es posible que cada unidad funcional consista en un proceso relativamente sencillo comparado con el proceso global, reduciendo la posibilidad de errores en la programación y permitiendo el empleo de unidades de control más sencillas y por lo tanto más económicas. Si existiera un fallo en las diferentes unidades de control esto no implicaría que el proceso global deba detenerse ni tampoco interrumpir los otros procesos. La desventaja de esta metodología es que se necesita realizar un estudio de implantación previo, ya que se deben identificar los procesos autónomos, asignar elementos a cada proceso y diseñar el modelo de intercomunicación para responder a las necesidades del proceso planteado (Muñoz, 2009).

1.3.3 Control Hibrido.

El control híbrido no está muy bien definido ya que este tipo de gestión puede considerarse a cualquier estrategia de distribución de elementos de control que esté en medio del control distribuido y el control centralizado. En ciertas ocasiones no es sencillo separar los procesos de manera completamente autónoma, por lo que se debe recurrir a la gestión de varios procesos desde una misma unidad de control, pues la complejidad de la separación es mayor que la complejidad que supone su gestión continua.

Una estrategia de este tipo también conduce a una gestión estructurada, de modo que existen elementos de control de nivel superior que supervisan e intercomunican los procesos autónomos más sencillos, siendo los encargados de gestionar la información. Para este tipo de gestión también es necesario el uso de redes de comunicación (Lontorfos, Fairbanks, Watkins, & Robinson, 2015).

1.4 Pirámide de automatización.

La automatización de los procesos productivos es uno de los aspectos que más ha evolucionado en la industria desde sus comienzos. La integración de tecnologías clásicas como la mecánica y la electricidad con otras más modernas como la electrónica, informática, telecomunicaciones, están haciendo posible esta evolución.

Esta integración de tecnologías queda representada en la llamada "pirámide de automatización", que recoge los cuatro niveles tecnológicos que se pueden encontrar en un entorno industrial. Las tecnologías se relacionan entre sí, tanto dentro de cada

nivel como entre los distintos niveles a través de los diferentes estándares de comunicaciones industriales (Comind, 2016).

Figura 1.7: Pirámide de automatización.
Fuente: (Comind, 2016).

1.4.1 Nivel de entrada/salida

Llamado también nivel de instrumentación. Está formado por los elementos de medida (sensores) y mando (actuadores), distribuidos en una línea de producción. Estos elementos están relacionados con el proceso productivo ya que los actuadores son los que se encarga de ejecutar las órdenes de los elementos de control para modificar el proceso productivo (por ejemplo un motor que sirva para mezclar) y los sensores miden las variables en el proceso de producción (ejemplo la temperatura en la línea de producción). Los sensores y actuadores suelen ser dispositivos que necesitan ser controlados por otros elementos.

1.4.2 Nivel de campo y proceso

En este nivel se encuentran los elementos capaces de gestionar los actuadores y sensores del nivel de célula, los mismos que pueden ser autómatas programables o equipos de aplicación específica basados en microcontroladores. Estos dispositivos permiten que los actuadores y sensores funcionen conjuntamente para que sean capaces de realizar el proceso industrial deseado. Son dispositivos programables, de manera que es posible ajustar y personalizar su funcionamiento según las necesidades de cada caso. Los dispositivos de este nivel de control en conjunto con los de nivel inferior de acción/sensado poseen entidad suficiente como para realizar procesos

productivos por sí mismos. Gran cantidad de procesos industriales están basados exclusivamente en estos dos niveles, de modo que un proceso productivo completo se desglosa en subprocesos de este tipo sin que exista un intercambio de información entre ellos.

A pesar de tratarse de proceso aislados, esto no significa que no se empleen buses de comunicación. Ya que para procesos que requieran de un gran número de sensores y actuadores, es recomendable la utilización de buses de campo para leer el estado de los sensores, proporcionar señales de control a los actuadores y conectar diferentes autómatas programables para compartir información acerca de la marcha del proceso completo.

Es importante que estos dispositivos tengan buenas características de interconexión para ser enlazados con el nivel superior, generalmente a través de buses de campo (Muñoz, 2009).

1.4.3 Nivel de control

Es posible monitorizar los dispositivos de control existentes en planta si tenemos un sistema de comunicación adecuado capaz de comunicar estos elementos con otro tipo de dispositivos no dedicados al control sino para la gestión y supervisión, y que regularmente pueden ser computadores o sistemas de visualización como pantallas industriales. En este nivel podemos visualizar cómo se llevan a cabo los procesos de planta, y por medio de entornos SCADA tener una "imagen virtual de la planta" de modo que ésta se puede recorrer de manera detallada, o también mediante pantallas de resumen, visualizar en un "panel virtual" donde se muestren posibles alarmas, fallos o alteraciones en cualquiera de los procesos que se llevan a cabo. Mediante este tipo de acciones podemos disponer de un acceso inmediato a cada uno de los sectores de la planta. Para esto, es esencial la conexión con el nivel de control mediante buses de campo de altas prestaciones, ya que la transmisión de información puede contener importantes cantidades de datos y también es necesaria la conexión con un gran número de elementos de control. Para las diferentes tareas que se realiza en un proceso industrial se utiliza un autómata, para esto el sistema de supervisión debe ser capaz de acceder al estado de cada uno de ellos, visualizar el proceso que lleva a cabo, y de manera global, tener información de cómo están trabajando cada uno de ellos, así como poder acceder a informes que genera el autómata. A medida que pasa el tiempo

también es posible modificar los procesos productivos desde los computadores de supervisión.

1.4.4 Nivel de Gestión (nivel de fábrica)

El nivel de gestión estará constituido principalmente por computadores ya que se encuentra más alejado de los procesos productivos. En este nivel no es relevante el estado y la supervisión de los procesos de planta, lo que si adquiere importancia en este nivel es toda la información relativa a la producción y su gestión asociada, a través del nivel de supervisión es posible obtener información de todos los niveles inferiores de una o varias plantas. Un ejemplo de la utilidad de la comunicación de los niveles inferiores con el nivel de gestión es la obtención de información en este nivel acerca de la materia prima consumida, la producción realizada, los tiempos de producción, niveles de almacenado de productos finales, etc. Con la información obtenida los gestores de la empresa pueden extraer estadísticas acerca de los costos de fabricación, rendimiento de la planta, estrategias de ventas para liberar posibles excesos de producto almacenado, y en general, disponer de datos que permitan a los niveles directivos la toma de decisiones que conduzcan a una mejor optimización de la planta, todo esto gracias al acceso a los datos de fabricación de manera rápida y flexible. Las comunicaciones con este nivel de la pirámide industrial ya no necesitan ser del tipo estrictamente industrial, sino que los datos que se transmiten son informes que pueden tener un tamaño medio, por lo que se emplean redes de comunicación menos costosas como redes Ethernet que se adaptan mejor al tipo de datos que se desean transmitir y que permiten una comunicación eficaz entre los diferentes computadores del mismo nivel de gestión (Comind, 2016).

1.5 Tipos de protocolos de comunicación industrial

1.5.1 Modbus

Figura 1.8: Logo Modbus.
Fuente: (Organization, 2017).

Fue desarrollado por Modicon en 1979 y es utilizado para establecer conexiones Maestro-Esclavo y Cliente-Servidor, transmite señales digitales y analógicas entre ellos, el protocolo utiliza el estándar RS-232, que define las características físicas de la conexión.

En comunicación Maestro-Esclavo, el maestro es un panel de operador o un nodo central mientras que un esclavo puede ser un autómata programable y su comunicación puede ser punto a punto con un único esclavo o mensaje general (*broadcast*) (MODICON, 1996).

1.5.2 MAP/TOP

General Motor en 1980 desarrolla un protocolo de comunicaciones industriales capaz de alcanzar altas tasas de comunicación de información llamado MAP[6], basado en la estructura de siete capas propuesto por la ISO[7].

El protocolo TOP[8] implementado por Boeing fue creado para eliminar barreras de comunicación en sus oficinas, intentando la integración de sistemas informáticos, este protocolo tiene similitudes con MAP, debido a esto los dos grupos se fusionaron años más tarde (Muñoz, 2009).

1.5.3 Profibus

Figura 1.9: Logo ProfiBus.
Fuente: (Profibus, 2016).

Fue creado en 1989 por un consorcio de cuatro empresas y siete universidades, FMS[9] es un protocolo orientado al intercambio de grandes cantidades de datos autómatas.

[6] MAP: *Manufacturing Automation Protocol*

[7] ISO: *International Standars Organization*

[8] TOP: *Technical Office Protocol*

[9] FMS: *Fieldbus Message specification*

Es un protocolo que proporciona una solución al cableado sencillo para tareas de comunicación maestro-esclavo, Actualmente está introducido en todos los niveles de automatización, desde el nivel de entrada/salida hasta sistemas que gestionan grandes cantidades de datos (Penin A. R., 2012).

1.5.4 Fieldbus Foundation

Figura 1.10: Logo Fieldbus.
Fuente: (Foundation, 2015).

Es una organización dedicada a crear un bus de campo único y abierto, inicio en 1994 a partir de la *WorldFIP north América* y de *Interoperable Systems Project*, está conformada por más de 350 fabricantes y usuarios.
Fieldbus implementa tecnologías de comunicación de forma que pueden soportar aplicaciones críticas, es el único protocolo de bus de campo digital desarrollado para el cumplimiento de las especificaciones SP50 (Penin A. R., Sistemas SCADA, 2007).

1.5.5 ASi

Figura 1.11: Logo ASi.
Fuente: (ASinterface, 2017).

Llamado también AS[10]- interface es un estándar internacional, tiene como finalidad monitorear y uniformizar el nivel de control, es un sistema diseñado para transmitir alimentación y datos con distancias de hasta 100m, es un sustituto al cableado

[10] As: *Asisociation for promoting Interfaces with bus capability for binary Actuators and Sensors*

tradicional, siendo este digital, proporcionando un nivel de flexibilidad, fiabilidad y ahorro superiores frente al sistema clásico (Becerra, 2002).

1.5.6 LonWorks

Figura 1.12: Logo LonWorks.
Fuente: (lonworks.es, 2015).

Es un conjunto de protocolos llamados LONtalk, consiste en una serie de protocolos que permiten la comunicación inteligente entre los dispositivos de la red, este protocolo fue incluido en el estándar ANSI/EIA[11] 709.1 en 1999.

LONworks[12] emplea como concepto básico para definir su red como una "red de control", en contraste con las redes de datos que tradicionalmente se conocen, la comunicación se hace entre nodos punto a punto o bien Maestro-Esclavo (Muñoz, 2009).

1.5.7 Interbus

Figura 1.13: Logo Interbus.
Fuente: (Interbus Organization, s.f.).

Fue desarrollado por la sociedad *Phoenix Contac* como un sistema de entradas y salidas numéricas de hasta 512 esclavos, dedicado para aplicaciones estándar de entradas y salidas distribuidas.

---

[11] ANSI/EIA: *American National Standards Institute/Electronic Industries Alliance standard*

[12] LONworks: *Local operating networks*

El maestro se comporta como una tarjeta de entradas y salidas del PLC, utiliza una topología estrella, todos los nodos son activos, las señales de ida y vuelta son integradas en la misma línea (Penin A. R., 2012).

1.5.8 DeviceNet

Figura 1.14: Logo DeviceNet.
Fuente: (SMART, 2017).

Está basada en el protocolo CAN, fue desarrollada en los años 90 e integrado posteriormente por ODVA[13], es una red digital de tipo abierto, flexible en su implementación y bajo costo. Es una tecnología diseñada para satisfacer las exigencias de fiabilidad requeridas por los ambientes industriales. DeviceNet permite que un fabricante añada configuraciones exclusivas de sus productos además de las mínimas requeridas por el protocolo (Muñoz, 2009).

1.5.9 Hart

Figura 1.15: Logo Hart.
Fuente: (SMART, 2017).

Fue desarrollado por Rosemount en los años 80 como un protocolo abierto. En 1993 se crea Hart[14] con la finalidad de mantener la propiedad de su tecnología y asegurar la accesibilidad entre los sectores industriales.

Es un protocolo muy difundido en la industria de procesos, los módulos de este tipo se agrupan en el HARD *User Group* y así garantizan el soporte técnico gracias a la HART *Communication Foundation* (Penin A. R., Sistemas SCADA, 2007).

[13] ODVA; Open Devicenet vendor Association

[14] Hart: *Highway Addressable Remote Transducer.*

1.5.10 ControlNet

Figura 1.16: Logo ControlNET.
Fuente: (Networks, 2015).

Es una red de comunicaciones industrial orientada en tiempo real gracias a su elevada velocidad de transferencia, el campo de aplicación es toda red que requiera entradas y salidas digitales como líneas automáticas de ensamblado, tratamiento de aguas, procesos alimenticios, industria farmacéutica y transporte de productos (Becerra, 2002).

1.5.11 WorldFIP

Figura 1. 17: Logo WorldFIP.
Fuente: (worldfip, 2017).

Es una norma de bus de campo que propone ser un sistema de gestión en una base de datos industrial, cumple tres principales funciones: Productor de datos, Consumidor de datos, distribuidor.

WorldFip es un protocolo para operaciones de proceso, su principal aplicación se da en la industria automovilística, la simplicidad inherente del protocolo ofrece al usuario una entrega garantizada de variables de tiempo crítico y le brinda la posibilidad de transferir archivos de datos en el mismo bus (Penin A. R., 2012).

1.5.12 Ethernet IP

Figura 1.18: Logo Ethernet IP.
Fuente: (Networks, 2015).

Es un protocolo basado en DeviceNet y ControlNet, sus principales características son: su versatilidad de interconexión en cualquier lugar, su flexibilidad ya que en la actualidad cualquier ordenador tiene un puerto de conexión a red local (Ethernet) y su bajo costo, utiliza una topología en estrella (Penin A. R., 2012).

1.5.13 CAN

Figura 1.19: Logo Cia.
Fuente: (CANopen, 2017).

El protocolo CAN fue introducido por Robert Bosh en 1986 como una solución al problema del cableado en la industria automovilística, para el desarrollo del bus se tiene la colaboración de Intel y Mercedes-Benz, posteriormente en 1992 se funda el grupo Cia[15] presentando mejoras y extensibilidad para otras industrias (Penin A. R., 2012).

[15] Cia: Can *in Automation*.

1.5.14 Cuadro comparativo de los protocolos de comunicación industriales

Tabla 1.1: Topología y ventajas de protocolos de comunicaciones industriales.

Protocolo	**Topología**	**Ventajas**
Modbus	Lineal, estrella, árbol, red con segmentos.	Conexión sencilla a sistemas Modicon, adecuado para cantidades de datos pequeñas (menores o iguales a 255 *Bytes*) y transferencia de datos con acuse.
Profibus	Lineal, estrella, anillo.	Transmite pequeñas cantidades de datos, cubre necesidades en tiempo real, fácil configuración
Fieldbus	Estrella.	Costo de instalación y mantenimiento reducido, mejoramiento de desempeño.
Asi	Bus, estrella, anillo.	Montaje sencillo y configuración sencilla, derivación económica del bus sin repetidores, sencilla posibilidad de ampliación.
Lonworks	Bus, anillo, estrella, lazo.	Diseñado para un amplio rango de aplicaciones.
Interbus	Segmentado	Costos reducidos de instalación, fácil detección de errores.
DeviceNet	Troncal, puntual, bifurcación.	Reducción en el cableado de la planta, capacidad de tener un puente sobre las redes de un nivel más alto.
Hart	Bus, lineal.	Permite soportar hasta 256 variables, entrega una alternativa económica en comunicación digital.
ControlNet	Árbol, estrella, bus.	Permite que los reguladores múltiples controles I/O en el mismo conductor.
EthernetIP	Bus, estrella, malla, cadena.	Aprovechamiento del ancho de banda, la conexión del cable Ethernet es más segura.
CAN	Bus lineal.	Minimiza el tiempo fuera de servicio del bus y aumenta al máximo el uso eficaz de ancho de banda disponible.

1.6 Modbus

1.6.1 Introducción

Es un protocolo utilizado para establecer conexiones cliente-servidor y maestro-esclavo, transmite señales digitales y analógicas, fue diseñado para trabajar con equipos como PLC, microcontroladores, computadoras, motores, sensores y otros dispositivos de entrada/salida.

La red Modbus emplea el concepto de bus de campo de control, ésta utiliza los niveles 1, 2 y 7 del modelo OSI, que corresponden al nivel físico, enlace y aplicación respectivamente. Su topología es lineal en donde existe un maestro el cual controla

acceso al medio y monitoriza el funcionamiento de la red, y uno o más dispositivos que actúan como esclavos como se muestra en la figura 1.20.

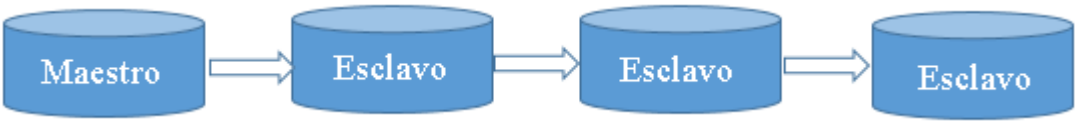


Figura 1.20: Topología bus lineal de ModBus.

La comunicación se da de forma serial asíncrono bajo los estándares RS-232 o RS-485 para enlaces semi-dúplex y RS-422 para enlace dúplex. Este protocolo tiene dos modos de transmisión el ASCII[16] y RTU[17] para la intercomunicación de mensajes entre diferentes dispositivos que conforman la red, estos mensajes conocidos como tramas contienen los datos necesarios para reconocer el origen y objetivo de cada mensaje puesto en el bus por alguno de los dispositivos, la longitud de cada trama depende del modo de transmisión, está acotada por un máximo de 256 caracteres. Para la comunicación el usuario selecciona el modo de transmisión deseado junto con los parámetros de velocidad de transmisión, paridad, número de bits de parada, etc. (Gelvez, 2002).

Este protocolo maneja dos tipos de datos, *bits* individuales y palabras de 16 *bits*, las palabras de 16 *bits* corresponden a registros de entrada y salida cuyos estados indican un valor analógico, los *bits* individuales corresponden a entradas o salidas discretas con estados de encendido/apagado (Olaya, Barandica Lopez, & Guerrero Moreno, 2004).

Tabla 1.2: Datos en un dispositivo de una red Modbus.

Sector	**Formato**	**Tipo de acceso**	**Comentario**
Salidas discretas	Bits individuales	Lectura-escritura	Modificables por un programa de aplicación
Entradas discretas	Bits individuales	Solo lectura	Suministrados por un sistema E/S

[16] ASCII: *American Standard Code for Information Interchange.*

[17] RTU: *Remote Terminal Unit.*

Registro de entrada	Palabras de 16 bits	Solo lectura	Suministrado por un sistema de E/S
Registro de salida	Palabras de 16 bits	Lectura-escritura	Modificables por un programa de aplicación

En la tabla 1.2 se muestran los cuatro tipos de datos que se pueden presentar en los controladores que tiene el protocolo Modbus, el formato en el que se encuentran dentro del dispositivo, el tipo de acceso y como pueden ser modificados o suministrados. Las entradas no tienen la posibilidad de ser cambiadas por un *software* de aplicación ya que estas hacen referencia a estados externos de los controladores.

1.6.2 Medios de transmisión

El protocolo de comunicación Modbus tiene dos modos de transmisión ASCII y RTU, el modo a utilizar será seleccionado por el usuario, junto con los parámetros de comunicación del puerto serie como: velocidad, paridad, baudios. Todos los dispositivos Modbus que estén conectados en red deben tener el mismo modo de transmisión y los mismos parámetros. La selección del modo ASCII o RTU tiene que ver únicamente con redes Modbus estándar, esto define los bits contenidos en los campos del mensaje transmitido en forma serial, también determina cómo se debe decodificar y empaquetar la información en los campos del mensaje.

1.6.2.1 ASCII

En este modo de transmisión cada 8 *bits* de un mensaje se envían como dos caracteres ASCII, la ventaja de esta modalidad es que se permiten tiempos muertos de hasta un segundo entre caracteres sin provocar errores y tiene una codificación hexadecimal.
Para un valor hexadecimal contenido en cada caracter ASCII del mensaje tenemos:

- 1 *bit* de inicio.
- 8 *bits* de datos (el menos significativo primero).
- 1 *bit* de paridad (0 *bit* si no hay paridad).
- 1 *bit* de parada con paridad (2 *bits* de parada sin paridad).
- Campo de verificación de error.
- Verificación de redundancia longitudinal (LRC) (Penin A. R., Sistemas SCADA, 2007).

1.6.2.2 RTU

Para el modo de transmisión RTU cada *byte* del mensaje contiene dos caracteres hexadecimales de 4 *bits*. Una de sus ventajas es la densidad de caracteres, más elevada de ASCII, que aumenta la tasa de transmisión manteniendo la velocidad, la codificación es binaria de 8 *bits* en hexadecimal, dos caracteres hexadecimales por cada *byte* de mensaje.

- 1 *bit* de inicio.
- 8 *bits* de datos (el menos significativo primero).
- 1 *bit* de paridad (0 *bit* si no hay paridad).
- 1 *stop bit* con paridad (2 *stop bits* sin paridad).
- Campo de verificación de error (*check field*).
- Verificación de error (*check field*).
- Verificación de redundancia cíclica (CRC).

Los mensajes comienzan tras un silencio de 3.5 caracteres, a continuación viene la dirección del dispositivo, después los elementos de la red monitorizan a la espera de un caracter de silencio, a continuación codifican el dato de dirección y para finalizar un intervalo similar marcará el fin del mensaje. De este modo se observa que un mensaje debe transmitirse de manera continua para no generar errores de transmisión (Penin A. R., Sistemas SCADA, 2007).

1.6.3 Tramas

Los intercambios de mensajes cumplen un ciclo de pregunta/respuesta en los dos modos de transmisión RTU y ASCII, esto se logra mediante tramas como muestra la figura 1.21.

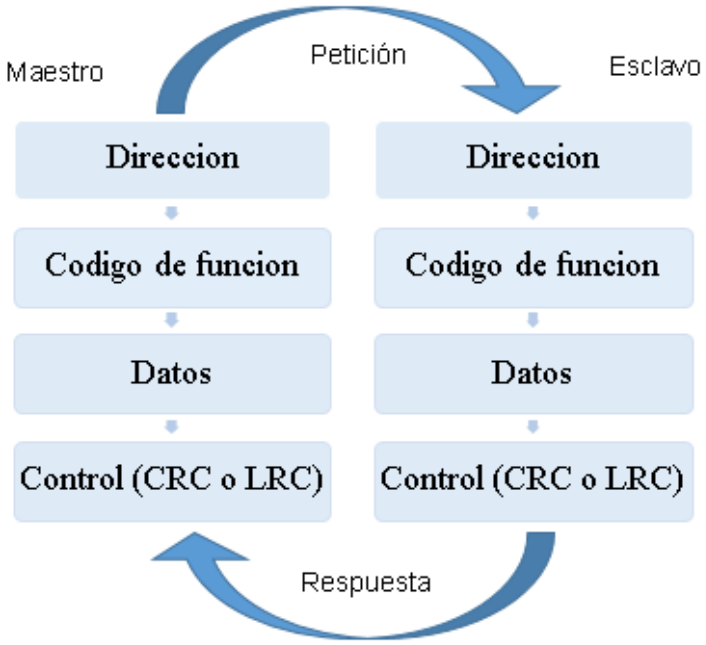


Figura 1.21: Tramas Modbus.

1.6.3.1 Campo de Dirección

Modbus es un protocolo multipunto esto quiere decir que el maestro puede comunicarse a cualquier esclavo utilizando la misma línea de transmisión, esta topología se conoce como tipo bus. Para que el maestro pueda reconocer a los diferentes esclavos estos deben tener una identificación única e irrepetible dentro de la red con la que se identificará el destino y el origen de los mensajes que sean puestos en el bus, si se llegaran a duplicar las identificaciones producirían colisiones en el bus o conflictos en la red que conlleven a un flujo de datos no confiable. La dirección de un dispositivo en la red debe estar, según el documento (MODICON, 1996) en el rango 1 a 63 con posibles valores que van desde el 01h hasta 3FH, esto con el fin de garantizar que los bits que conforman la trama lleguen con los niveles de tensión o corriente establecidos en los estándares de comunicación serial utilizados.

En caso de que el maestro se comunique a todos los esclavos de la red al mismo tiempo, el campo de dirección debe contener como dato un cero (00H), esta difusión es llamada *Broadcast*. Cuando una pregunta se hace por medio de una difusión ninguno de los esclavos debe enviar un mensaje de respuesta, el campo de dirección es utilizado por los esclavos para colocar la dirección y así ser identificado por el maestro en los mensajes de repuesta (Rondon & Arenas, 2011).

1.6.3.2 Campos de códigos de función

Cada función permite transmitir órdenes o datos a un esclavo, existen dos tipos básicos de órdenes:

- Ordenes de lectura/escritura de datos en los registros o en la memoria del esclavo.
- Órdenes de control (*RUN/STOP*), carga y descarga de programas, verificación de contadores, etc.

Tabla 1.3: Funciones básicas y códigos de operación ModBus.

Código	**Hex**	**Descripción**
0	00	Control de estaciones esclavas
1	01	Lectura de n *bits* de salida
2	02	Lectura de n *bits* de entrada
3	03	Lectura de n registros de entradas
4	04	Lectura de n registros de salidas
5	05	Escritura de un *bit*
6	06	Escritura de una palabra o registro
7	07	Lectura rápida de 8 *bits*
8	08	Diagnóstico
9	09	No utilizada
10	0A	No utilizada
11	0B	Solicitar contador de eventos de comunicaciones
12	0C	Solicitar diario de eventos de comunicaciones
13	0D	No utilizada
14	0E	No utilizada
15	0F	Escritura de n *bits*. Bobinas (*Coils*)
16	10	Escritura de n palabras. (*Holding Registers*)

Fuente: (MODICON, 1996).

La tabla 1.3 nos muestra las funciones básicas disponibles en el protocolo ModBus con sus códigos de operación. Cuando se utiliza el modo de transmisión ASCII, el campo de código de función de una trama contiene dos caracteres u ocho bits para el caso del modo de transmisión RTU. Los códigos válidos están en el rango decimal de

1 a 255. Si el maestro envía un mensaje de petición a un dispositivo esclavo, el código de función le dice al esclavo que tipo de acción debe ejecutar, algunos ejemplos de funciones son: leer o forzar los estados *ON/OFF* de un grupo de salidas discretas, leer o forzar el contenido de un grupo de registros, leer el estado de diagnóstico del esclavo. Para enviar su respuesta el esclavo usa el campo de código de función en el mensaje para indicar si es una respuesta normal o si ha ocurrido alguna excepción (respuesta de excepción), si la respuesta es normal, el esclavo debe reenviar el código de función recibido en la petición, en cambio, para una respuesta de excepción el esclavo proporciona un código que es equivalente al código de función original con su bit más significativo en 1 lógico.Por ejemplo si un maestro envía un mensaje al esclavo para leer n *bits* de entrada, tendría el siguiente código de función en el modo de transmisión RTU:

0000 0010 (Hexadecimal 02).

Si el esclavo puede ejecutar la acción que el maestro le solicita, el código de función enviado como mensaje de respuesta es el mismo, si ocurre una excepción el contenido del campo de código de función será:

1000 0010 (Hexadecimal 82).

Para una respuesta de excepción, además de la modificación del código de función anteriormente mencionada, el esclavo coloca un código en el campo de datos del mensaje de respuesta, el cual informa al maestro que tipo de error se produjo o la razón de la excepción (MODICON, 1996).

1.6.3.2.1 Respuestas de Excepción

Durante el ciclo consulta-respuesta pueden ocurrir diversos eventos que obligan al esclavo a enviar un mensaje de excepción, los mensajes de excepción no serán enviados si existen errores de comunicación como son errores de paridad en el LRC [18]o CRC[19].

Los mensajes de excepción se presentan si ocurre alguno de los siguientes eventos:

- Al enviar el mensaje de consulta se le solicita el esclavo que ejecute una función que no soporta.

[18] LRC: *Longitudinal Redundancy Check.*
[19] CRC: *Cyclical Redundancy Check.*

- Cuando se le solicita un *bit* o un dato inexistente.
- Si el número de datos solicitado excede el máximo del esclavo.
- Cuando el esclavo tiene dificultades o detecta errores al ejecutar una función.
- Si el esclavo está ocupado.
- Si el esclavo requiere de un tiempo mayor al *timeout* para ejecutar la función solicitada.
- Si se detecta error en la ejecución de una función enviada en un programa.
- Cuando se detectan errores en una memoria extendida.

Cada evento tiene un código con el que el maestro determinará el origen del mensaje de excepción. En la tabla 1.4 se muestran los códigos de error definidos en el protocolo ModBus.

Tabla 1.4: Códigos de Excepción.

Código	**Nombre**	**Significado**
01	Función Ilegal	El código de función recibido en la consulta no es soportada por el esclavo.
02	Dirección de dato ilegal	La dirección de datos en la consulta no es una dirección válida para el esclavo.
03	Valor de datos ilegal	Una dirección enviada dentro del campo de datos en la consulta no es una dirección válida para el esclavo.
04	Falló el dispositivo esclavo	Un error desconocido ocurrió mientras el esclavo intentaba ejecutar la función solicitada en la consulta.
05	Reconocimiento	El esclavo ha aceptado la consulta y está procesándola, pero requiere de mayor tiempo para ejecutarla. Esta respuesta es enviada para evitar un error por *timeout* en el maestro.
06	Dispositivo esclavo ocupado	El esclavo está ejecutando un comando de programa que requiere de mucho tiempo.
07	Reconocimiento negativo	El esclavo no puede ejecutar la función recibida en la consulta. Este código es enviado por una consulta de programa infructuosa que usa los códigos de función 13 o 14 decimal. El maestro debe hacer una consulta de diagnóstico o información de error del esclavo.
08	Error de paridad de memoria	El esclavo intenta leer una memoria extendida, pero detecta un error de paridad en esta. El maestro puede enviar la consulta nuevamente, pero algún servicio puede ser necesario en el esclavo.

Fuente: (MODICON, 1996).

1.6.3.3 Campos de Datos

El campo de datos se construye usando grupos de dígitos hexadecimales, en el rango de 00 hasta FF hexadecimal a partir de un caracter RTU o de un par de caracteres ASCII, de acuerdo con el modo de transmisión serie de la red.
Este campo entrega información adicional que el esclavo debe usar para tomar la acción definida por el código de función, también puede incluir ítems como son direcciones iniciales de entrada o salida, direcciones iniciales de registros de entradas o salidas, el número de datos a leer, etc.
Una solicitud de un dispositivo maestro a esclavo con su registro de evento de comunicaciones (código de función 0B hexadecimal), el esclavo no requiere ninguna información adicional por lo que el campo de datos puede ser inexistente o de longitud cero, el código de función solo especifica la acción (MODICON, 1996).

1.6.3.4 Campo de chequeo de errores

Este es el último campo de la trama, permite al maestro y esclavo detectar errores de transmisión, estos errores frecuentemente se ocasionan por ruidos eléctricos o interferencias de otra naturaleza, esto produce que el mensaje sufra alguna modificación mientras se está transmitiendo. En este campo se asegura que los dispositivos receptores no ejecuten ninguna acción incorrecta si se detectara dicho error.
Para las redes Modbus existe un tipo de método de chequeo de error dependiendo del modo de comunicación utilizado.
Para modo ASCII se utiliza el chequeo de redundancia longitudinal (LRC) que se lleva a cabo con los contenidos de mensaje.
Para el modo RTU se utiliza el chequeo de Redundancia Cíclica (CRC), el campo de chequeo de error contiene un valor de 16 *bits* implementando como dos *bytes* de 8 *bits*.
El campo de CRC se añade al mensaje como el último campo del mensaje. Cuando esto se hace, se añade primero el *byte* de menos significativo del campo, seguido por el *byte* de más significativo (MODICON, 1996).

1.6.3.4.1 CRC

El campo de corrección de errores contiene un valor binario de 16 *bits*, para detectar un error en la transmisión se tiene el siguiente proceso: el CRC es calculado por el

dispositivo que envía, el dispositivo receptor recalcula un CRC con los datos del mensaje y lo compara con el valor que se envió, si los dos valores no son iguales, se interpreta como un error en la transmisión. Se necesita de 6 pasos para calcular el CRC de un mensaje:

Paso 1.- Se carga un registro de 16 *bits* con FFFF hex[20]. Llamarlo registro CRC.

Paso 2.- Se realiza la operación XOR del primer *byte* del mensaje con el *byte* de orden bajo del registro CRC de 16 *bits*, poniendo el resultado en el registro CRC.

Paso 3.- Se desplaza el registro CRC un *bit* a la derecha, llenando con cero el MSB[21]. Se examina el *bit* de acarreo resultado del desplazamiento a la derecha.

Paso 4.- Si el acarreo es 0, repetir el Paso 3. Si el acarreo es 1, realizar la operación XOR del registro CRC con el valor del polinomio A001 hex.

Paso 5.- Repetir los pasos 3 y 4 hasta que se haya llevado al cabo ocho desplazamientos. Cuando esto se haya realizado, se habrá procesado un *byte* completo.

Paso 6.- Repetir los pasos 2 hasta 5 para el siguiente *byte* del mensaje. Continuar haciendo esta operación hasta que todos los *bytes* del mensaje hayan sido procesados.

Ejemplo 1 cálculo mensaje CRC

Datos enviados $P(x)= x^7+x^5+x^4+x^2+x^1+x^0$ o 10110111

CRC $G(x)=x^5+x^4x^1+x^0$ o 110011

Primero P(x) es multiplicado por el número de bits en el código CRC, 5.

$$x^5 (x^7+x^5+x^4+x^2+x^1+x^0)= x^{12}+x^{10}+x^9+x^7+x^6+x^5 = 1011011100000$$

Dividir para el CRC mediante XOR

1011011100000
110011
0111101
110011
0 0111010
110011
00100100
110011
0101110
110011

[20] hex: hexadecimal
[21] MSB: *Most Significant Bit* (*bit* mas significativo)

0111010
110011
01001 = Resto

Resto 01001 significa $R(x) = 0x^4+1x^3+0x^2+0x+1$

Se transmite entonces $T(x)= x^k P(x) + R(x)$ o sea 1011011101001

Si el receptor recibe 1011011101001, al dividir TR(x) por G(x) el resto será cero indicando que se recibió correctamente la información (Petroni, 2003).

1.6.4 Descripción de las funciones Modbus

Todas las direcciones en los mensajes Modbus son referenciadas a cero. La primera unidad de cada tipo de dato es direccionada como la número cero. Por ejemplo, la bobina número uno en un controlador programable es direccionada como bobina 0000 en el campo de dirección de un mensaje Modbus.
La bobina 150 decimal es direccionada como bobina 0095 hex (149 decimal).
El registro de salida 40001 es direccionado como registro 0000 en el campo de direcciones de un mensaje Modbus. El campo código de función específica una operación sobre un 'registro de salida'. Por lo tanto la referencia '4XXXX' está implícita (MODICON, 1996).
En la tabla 1.5 se muestran las referencias a los diferentes tipos de datos que se encuentran en el esclavo.

Tabla 1.5: Representación de los tipos de datos.

Referencia	Representación
0X	Bobinas
1X	Entradas Digitales
4X	Registros de retención

1.6.4.1 Leer Estados de Bobinas (01)

Esta función permite al maestro preguntar sobre el estado de las salidas discretas de un dispositivo remoto. Cuando se utiliza la palabra bobinas (*Coils*) se hace referencia a las salidas discretas de los controladores que actúan en una red Modbus, estas salidas tienen un estado binario *ON/OFF*. Las bobinas tienen referencia 0X. La transmisión tipo *broadcast* no está habilitada para esta función.

En la tabla 1.6 se muestra un mensaje donde se pide al esclavo 5 que envíe el estado de las bobinas de la 10 a la 22. En la tabla se observa que el maestro indica el número de la bobina a partir de la cual se debe empezar a enviar los estados de las bobinas, en los campos dirección de inicio; esto da como resultado el numero hexadecimal 0009 que hace referencia a la bobina 10.

En los campos número de salidas, se encuentra la parte alta y baja respectivamente de la cantidad de bobinas de las cuales se va a leer el estado. En este ejemplo el maestro pide el estado de 13 bobinas (00 0E Hex) a partir de la bobina 10 (MODICON, 1996).

Tabla 1.6: Leer estado de entrada (Consulta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	05
Función	01
Dirección de inicio (Parte Alta)	00
Dirección de inicio (Parte Baja)	09
Número de salidas (Parte Alta)	00
Número de salidas (Parte Baja)	0E
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

En el mensaje de respuesta el esclavo envía el estado de cada bobina empaquetando en bloques de ocho *bits* el estado de ocho bobinas consecutivas; representando una bobina por cada *bit* del campo de datos. 1 = *ON*, 0 = *OFF*.

Si la cantidad requerida de bobinas no son múltiplos de ocho, los *bits* restantes en el *byte* final de datos se llenarán con ceros hacia el MSB. En la tabla 1.7 se muestra una posible respuesta a la petición de la tabla 1.6.

Tabla 1.7: Leer estado de bobinas (Respuestas).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	15
Función	01
Contador de *Bytes*	05
Datos (Bobinas 27-20)	CD
Datos (Bobinas 35-28)	6B
Datos (Bobinas 43-36)	B2
Datos (Bobinas 51-44)	0E
Datos (Bobinas 56-52)	1B
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

El campo contador de *bytes se* especifica la cantidad de datos que el esclavo envía; en este ejemplo se tiene que el esclavo envía 5 *bytes* de datos, en los cuales está contenido el estado de todas las bobinas solicitadas por el maestro.
El estado de las bobinas 27 a la 20 se muestra como CD en hex, o binario 1100 1101. La bobina 27 es el MSB de este *byte*, y la bobina 20 es el LSB[22]. De izquierda a derecha, el estado de las bobinas 27 a la 20 es ON-ON-OFF-OFF-ON-ON-OFF-ON.

Por convención, los *bits* dentro de un *byte* se muestran con el MSB a la izquierda, y el LSB a la derecha. El siguiente *byte* tiene las bobinas 35 a la 28, de izquierda a derecha (MODICON, 1996).

1.6.4.2 Leer estados de las entradas (02)

Esta función permite al maestro preguntar sobre el estado *ON/OFF* de las entradas discretas de un dispositivo remoto, las entradas al igual que las salidas discretas se

[22] *LSB: Least Significant Bit* (bit menos significativo)

direccionan a partir del cero. La referencia de este tipo de datos es la 1X. La difusión *broadcast* no está habilitada para esta función.

Tabla 1.8: Leer estado de entrada (Consulta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	02
Dirección de inicio (Parte Alta)	00
Dirección de inicio (Parte Baja)	C4
Número de salidas (Parte Alta)	00
Número de salidas (Parte Baja)	16
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

En la tabla 1.8 se muestra una petición donde se le pide al esclavo 11 que envíe el estado de las entradas de la 197 a la 218. En esta petición se observa que el maestro indica el número de entrada a partir de la cual se debe empezar a enviar los estados de cada entrada, esto da como resultado el número hexadecimal 00C4 que hace referencia a la entrada 197. Se puede observar que C4 corresponde al número 196 en decimal.

En los campos número de salidas se encuentra la cantidad de entradas de las que se requiere conocer el estado. En este ejemplo el maestro pide el estado de la entrada (00 16 Hex) a partir de la entrada 00 C4 Hex.

Tabla 1.9: Leer estado de entrada (Respuesta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	02
Contador de *Bytes*	03
Datos (Entradas 10204-10197)	AC
Datos (Entradas 10212-10205)	DB
Datos (Entradas 10218-10213)	35
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

En la tabla 1.9 se muestra una respuesta a esta petición, la forma en que se ordenan los estados para ser enviado en el mensaje de respuesta es idéntico al utilizado para enviar el estado de las bobinas, en donde el LSB del primer *byte* se envía el estado de la primera entrada solicitada y a partir del cual empieza el llenado de los siguientes *bytes*.

El estado de las entradas 10204-10197 se representa como el *byte* de valor AC hex, o binario 1010 1100, la entrada 10204 es el MSB de este *byte*, y entrada 10197 es el LSB. De izquierda a derecha el estado de las entradas 10204 a 10197 es: ON-OFF-ON-OFF-ON-ON-OFF-OFF.

El estado de las entradas 10218 a 10213 es: ON-ON-OFF-ON-OFF-ON. Los dos *bits* más significativos están puestos a cero.

1.6.4.3 Leer los registros de salida (03)

Esta función permite conocer el estado binario de los registros de salida (*Holding Registers*) de un esclavo. Los registros tienen 16 *bits* de longitud y se empiezan a direccionar a partir del registro cero. Los registros de salida tienen referencia 4X. La difusión *broadcast* no está habilitada para esta función.

Tabla 1.10: Leer estado de los Registros de Retención (Consulta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	03
Dirección de inicio (Parte Alta)	00
Dirección de inicio (Parte Baja)	6B
Número de salidas (Parte Alta)	00
Número de salidas (Parte Baja)	03
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

Como se observa en la tabla 1.10 en el mensaje de petición se utilizan dos *bytes* para especificar el registro inicial en los campos dirección de inicio, en este ejemplo el registro inicial es el 40108 el cual se muestra como 00 (Hex) en la parte alta junto con 6B en la parte baja. Por otra parte en los campos número de registros se especifica cantidad de registros que se deben enviar en la respuesta, en este ejemplo se deben enviar tres registros a partir del 40108, es decir, los 40108 al 41110 del esclavo con dirección 17 (11 Hex).

Tabla 1.11: Leer estado de los Registros de Retención (Respuesta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	03
Contador de *Bytes*	06
Datos Parte Alta (Registro 40108)	02
Datos Parte Baja (Registro 40108)	2B
Datos Parte Alta (Registro 40109)	00
Datos Parte Baja (Registro 40109)	00
Datos Parte Alta (Registro 40110)	00
Datos Parte Baja (Registro 40110)	64
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

Un mensaje de respuesta para esta función puede ser el que se muestra en la tabla 1.11. En este ejemplo se observa que el contador de *bytes* contiene un 06, indicando que se enviarán seis *bytes* de datos en donde están contenidos el estado binario de los tres registros solicitados en la petición, se debe observar en esta figura que cada registro se envía utilizando dos *bytes*, el primero para la parte alta y un segundo para la parte baja. El contenido del registro 40108 se representa con dos *bytes* de valores 02 2B Hex. Los contenido de los registros 40109-40110 son 00 00 y 00 64 hex respectivamente.

1.6.4.4 Leer registros de entrada (04)

Esta función permite leer el estado de los registros de entrada (*Input Registers*) de un esclavo. Los registros de entrada tienen una longitud de 16 *bits* y se empiezan a direccionar a partir del registro cero, los registros de entrada tienen referencia 3X. La difusión *broadcast* no está habilitada para esta función.

Tabla 1.12: Leer registros de entrada (Consulta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	04
Dirección de inicio (Parte Alta)	00
Dirección de inicio (Parte Baja)	08
Número de registros (Parte Alta)	00
Número de registros (Parte Baja)	01
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

El mensaje de consulta especifica el registro inicial y cantidad de registros a leer, como se muestra en la tabla 1.12 el registro inicial se indica en los campos dirección de inicio y el número de registros que se deben enviar a partir del registro inicial están indicados en los campos número de registros.

Tabla 1.13: Leer registros de entrada (Respuesta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	04
Contador de *Bytes*	02
Datos Parte Alta (Registro 30009)	00
Datos Parte Baja (Registro 30009)	0A
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

Una respuesta a esta petición nos muestra la tabla 1.13. El contenido binario de cada registro se empaqueta en el mensaje utilizando dos *bytes*. El primer *byte* del registro contiene los ocho *bits* de mayor orden y el segundo contienen los ocho *bits* de menor orden, en el campo contador de *bytes* se encuentra indicado la cantidad de datos que corresponden a los campos de datos en la respuesta.

1.6.4.5 Forzar bobinas individuales (05)

Mediante esta función el maestro puede indicarle a un esclavo que establezca una única bobina (referencia 0X) ya sea en *ON* u *OFF*. Cuando el maestro envía una consulta general (difusión), se debe forzar la misma bobina en todos los esclavos conectados en la red. En este caso ningún esclavo debe enviar un mensaje de respuesta.

Tabla 1.14: Forzar una única bobina (Consulta y Respuesta).

Nombre del Campo	Ejemplo (Hex)
Dirección de esclavo	11
Función	05
Dirección de la bobina (Parte Alta)	00
Dirección de la bobina (Parte Baja)	AC
Dirección de la bobina a forzar (Parte Alta)	FF
Dirección de la bobina a forzar (Parte Baja)	00
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

Se observa un mensaje de consulta en la tabla 1.14, en ésta el maestro utiliza dos campos para especificar la parte alta y baja respectivamente de la referencia de la bobina a forzar. Las bobinas se direccionan comenzando en cero.

Los campos datos son utilizados por el maestro para indicar al esclavo en qué estado se debe poner la bobina referenciada, si en estos campos se coloca FF 00 hex (parte alta y baja) el esclavo debe colocar la bobina en un *ON* (1 lógico), si se coloca 00 00 Hex lo debe colocar en *OFF* (cero lógico). Cualquier otra combinación será errónea y no afectará el estado de la bobina; en cuyo caso el esclavo debe enviar una respuesta de excepción. Una respuesta normal es un mensaje igual a la consulta.

1.6.4.6 Prestablecer un único registro (06)

Mediante esta función el maestro puede poner en un determinado estado binario los *bits* que conforman un registro de salida (referencia 4X). Cuando esta petición se hace mediante una difusión todos los esclavos establecerán el mismo registro con el estado binario indicado por el maestro en la petición. En la tabla 1.15 se muestra lo que puede ser una petición con este código de función.

Tabla 1.15: Preestablecer un único registro (Consulta y Respuesta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	06
Dirección de registro (Parte Alta)	00
Dirección de registro (Parte Baja)	01
Datos (Parte Alta)	00
Datos (Parte Baja)	03
Comprobación de error (CRC o LRC)	--

Fuente (MODICON, 1996)

En los campos dirección de registro se indica la referencia del registro que se desea preestablecer y en los campos dato se especifica el estado binario al que debe ser establecido. La respuesta normal es el mismo mensaje enviado en la consulta y se devuelve después de que el registro ha sido modificado.

1.6.4.7 Forzar múltiples bobinas (0F)

Fuerza varias bobinas (referencias 0X) consecutivas ya sea *ON* u *OFF*. Cuando es una consulta general, la función fuerza las mismas bobinas en todos los esclavos conectados. El mensaje de consulta especifica las referencias de bobinas que se desean forzar. Las bobinas se direccionan comenzando en cero: La bobina 1 se direcciona como 0.

La petición de estado *ON/OFF* está especificada por el contenido del campo de información de la consulta. Un valor lógico "1" en el campo solicita que la bobina correspondiente pase a *ON*. Un valor lógico "0" solicita que la bobina pase a OFF. En la tabla 1.16 se muestra un ejemplo de solicitud para forzar una serie de diez bobinas, comenzando en la bobina 20 (direccionada como 19, o 13 hex) en el esclavo 17. El contenido de la consulta tiene dos *bytes*: CD 01 hex (1100 1101 0000 0001 binario). Los *bits* binarios corresponden a las bobinas de la siguiente forma:

Tabla 1.16: Orden bits referente a bobinas.

Bit	1	1	0	0	1	1	0	1	0	0	0	0	0	0	0	1
Bobina	27	26	25	24	23	22	21	20	-	-	-	-	-	-	29	28

Fuente: (MODICON, 1996).

El primer *byte* transmitido (CD hex) direcciona las bobinas 27-20, con el *bit* menos significativo se direcciona a la bobina más baja (20). El siguiente *byte* (01 hex) direcciona las bobinas 29-28, con el *bit* menos significativo se direcciona a la bobina más baja (28). Los *bits* no utilizados en el último *byte* de información deberán ser llenados con ceros.

Tabla 1.17: Forzar múltiples bobinas (Consulta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	0F
Dirección de Bobina (Parte Alta)	00
Dirección de Bobina (Parte Baja)	13
Numero de Bobinas (Parte Alta)	00
Numero de Bobinas (Parte Baja)	0A
Contador de *Bytes*	02
Datos a Forzar (Bobinas 27-20)	CD
Datos a Forzar (Bobinas 29-28)	01
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

La respuesta normal devuelve la dirección del esclavo, el código de operación, la dirección de inicio y la cantidad de bobinas forzadas.

Tabla 1.18: Forzar múltiples bobinas (Respuesta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	0F
Dirección de Bobina (Parte Alta)	00
Dirección de Bobina (Parte Baja)	13
Numero de Bobinas (Parte Alta)	00
Numero de Bobinas (Parte Baja)	0A
Comprobación de error (CRC o LRC)	--

Fuentes: (MODICON, 1996).

La tabla 1.18 muestra un ejemplo de respuesta a la consulta que se realizó en la tabla 1.17.

1.6.4.8 Forzar múltiples registros (10)

Esta función pre ajusta valores en una secuencia de registros de salida (referencias 4X). Cuando hay difusión, la función escribe sobre las mismas referencias de registro en todos los esclavos.

La tabla 1.19, presenta un ejemplo de una solicitud para pre-ajustar dos registros empezando en 40002 a 00 0A y 01 02 hex, en el dispositivo esclavo 17.

Tabla 1.19: Forzar múltiples registros (Consulta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	10
Dirección de Inicio (Parte Alta)	00
Dirección de Inicio (Parte Baja)	01
Número de Registros (Parte Alta)	00
Número de Registros (Parte Baja)	02
Contador de Bytes	04
Datos (Parte Alta)	00

Datos (Parte Baja)	0A
Datos (Parte Alta)	01
Datos (Parte Baja)	02
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

Tabla 1.20: Forzar múltiples registros (Respuesta).

Nombre del Campo	**Ejemplo (Hex)**
Dirección de esclavo	11
Función	10
Dirección de Inicio (Parte Alta)	00
Dirección de Inicio (Parte Baja)	01
Número de Registros (Parte Alta)	00
Número de Registros (Parte Baja)	02
Comprobación de error (CRC o LRC)	--

Fuente: (MODICON, 1996).

La respuesta normal devuelve la dirección del esclavo, el código de operación, la dirección de inicio y la cantidad de registros forzados. La tabla 1.20 muestra lo que puede ser una respuesta a la consulta de la tabla 1.19.

1.7 Conclusiones

Existen varios tipos de protocolos de comunicación industriales que pueden ser aplicados en diferentes entornos, debido a las ventajas que tiene el protocolo Modbus como el de ser de código y de fácil implementación, se utiliza en el desarrollo de este proyecto. La comunicación mediante Modbus permite establecer conexiones entre maestro-esclavo y cliente-servidor, las señales que se pueden transmitir son de tipo digital o analógico.

El maestro es el que controla la comunicación mientras que el o los esclavos reciben órdenes desde el maestro. La comunicación se da de forma serial asíncrono bajo los estándares RS-232 o RS-485 para enlaces semi-dúplex y RS-422 para enlace dúplex. Este protocolo tiene dos modos de transmisión el ASCII y RTU para la intercomunicación de mensajes entre diferentes dispositivos que conforman la red.

CAPÍTULO 2

IMPLEMENTACIÓN DEL SISTEMA DE COMUNICACIONES MEDIANTE EL PROTOCOLO MODBUS

2.1 Introducción.

El sistema de comunicación del proyecto contiene diferentes elementos de *hardware* y *software*, en los elementos físicos tenemos: PcDuino, módulos de radio frecuencia, Xbee, *Expander Pi*, y en *software* programas como Python y simuladores Modbus que serán explicados posteriormente.

En este capítulo se muestran las principales características de estos elementos así como también se indica paso a paso las configuraciones que se realizan para poder llevar a cabo una comunicación mediante el protocolo Modbus en los diferentes programas de simulación que existen para Modbus.

2.2 Tarjeta PcDuino 3

2.2.1 Características.

El PcDuino 3 es una mini PC de alto rendimiento y bajo costo. Funciona con sistemas operativos como Ubuntu, Linux y Android. Cuenta con una salida HDMI para TV o Monitor.

A continuación se muestran algunas características de *hardware* del PcDuinio3:

- CPU[23]: AllWinner A20 SoC, 1GHz ARM Cortex A7 Dual Core
- GPU[24]: OpenGL ES2.0, OpenVG 1.1, Mali 400 Dual Core
- DRAM[25]: 1GB
- Almacenamiento interno: 4GB Flash y *slot* de memoria SD[26] expandible hasta 32 GB.
- Salida de video: HDMI

[23] CPU: *Central Processing Unit* (Unidad Central de Procesamiento)
[24] GPU: *Graphics Processing Unit* (Unidad de Procesamiento Gráfico)
[25] DRAM: *Dynamic Random Access Memory* (Memoria Dinámica de Acceso Aleatorio)
[26] SD: *Secure Digital* (Seguridad Digital)

- Sistemas operativos compatibles: Ubuntu 12.04, Android 4.2
- Interfaz de red: Wi-Fi, Ethernet 10M/100Mbps
- Salida de audio: Interfaz de audio analógico de 3.5mm, Interfaz de audio digital estéreo I2S.
- 1 puerto USB
- Alimentación: 5V 2000mA

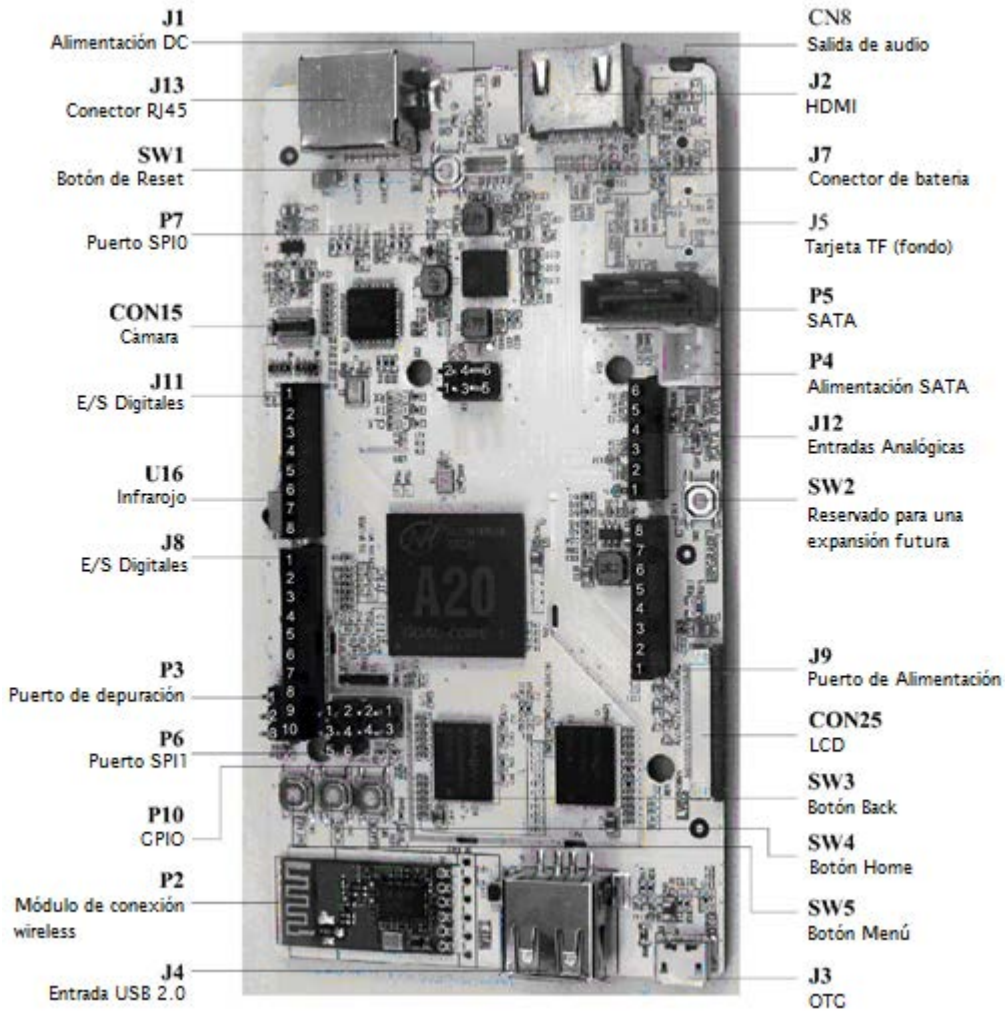


Figura 2.1: PcDuino 3.
Fuente: (LinkSprite, 2016).

El dispositivo es completamente de código abierto, por lo que los usuarios pueden programar sus propios proyectos sin ninguna restricción. Para ello, hay una gran cantidad de información detallada, diagramas de circuitos y ejemplos de programación en la página principal de pcDuino3 (ver figura 2.1).

2.2.2 Instalación de Ubuntu en PcDuino 3.

Para trabajar con el PcDuino 3 es necesario instalar el sistema operativo Ubuntu, la versión utilizada para este proyecto es Ubuntu 14. En la figura 2.2 se encuentran las opciones de descarga de los diferentes sistemas operativos para los modelos de PcDuino existentes.

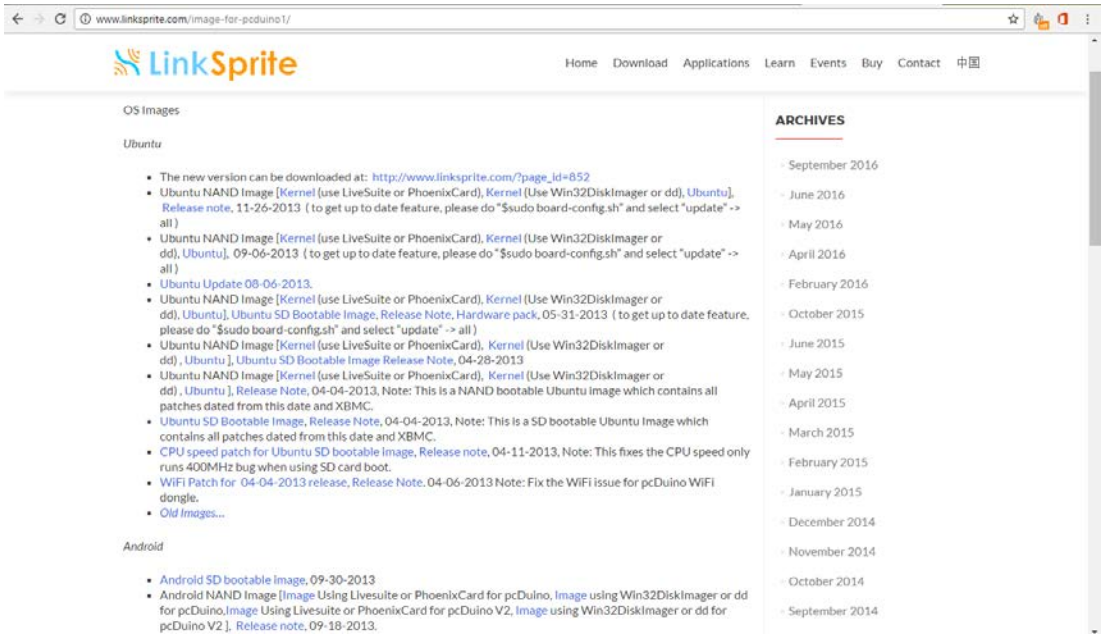


Figura 2.2: Página oficial de PcDuino.
Fuente: (LinkSprite, 2016).

El proceso de instalación se desarrolla en base al video tutorial que está en la página de PcDuino (LinkSprite, 2016), por medio de los siguientes pasos:

1. Se descarga el *software* "*Ubuntu NAND image*" para PcDuino 3 como se muestra en la figura 2.3.

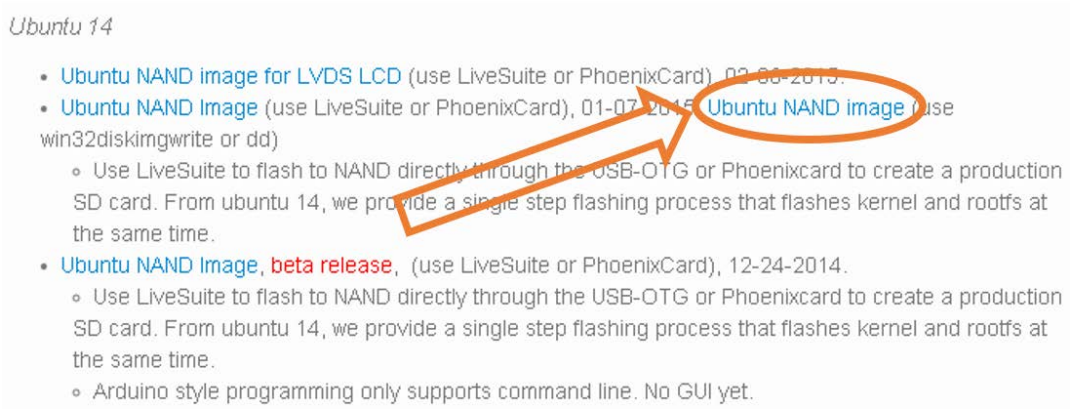


Figura 2.3: Imágenes de carga Ubuntu 14.
Fuente: (LinkSprite, 2016).

2. Se descarga la herramienta "Win32DiskImager", es utilizado para la carga de la imagen Ubuntu en una tarjeta SD. En la figura 2.4 se muestra el enlace de descarga.

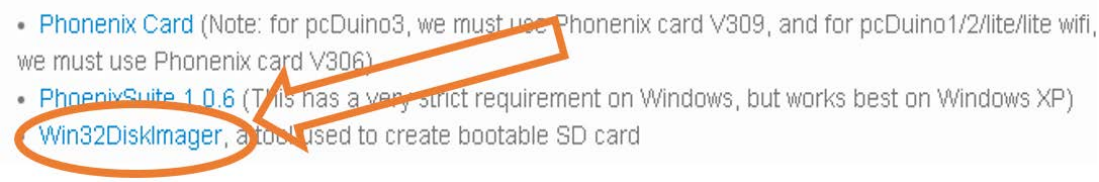


Figura 2.4: Herramientas de carga en PcDuino.
Fuente: (LinkSprite, 2016).

3. Se graba la imagen en una microSD mediante la herramienta "Win32DiskImager" como se muestra en la figura 2.5.

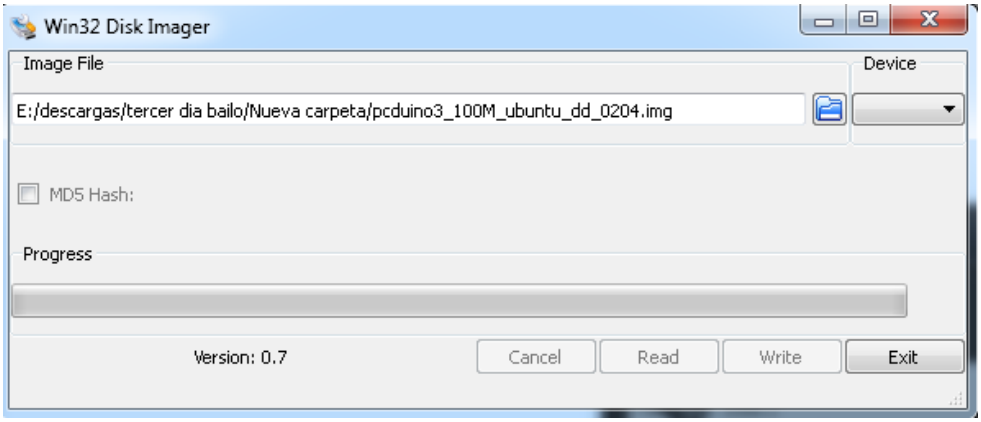


Figura 2.5: Programa Win32.

4. Al cargar la imagen se inserta la microSD en el PcDuino (ver figura2.6), a continuación se enciende el dispositivo y la instalación inicia automáticamente.

Figura 2.6: Imagen PcDuino y MicroSD.

5. Si no se presentan problemas durante la instalación, el sistema operativo se inicia y queda instalado como se observa en la figura 2.7.

Figura 2.7: Instalación de Ubuntu14 en el PcDuino.

2.2.3 Habilitar el puerto UART en el PcDuino

Para utilizar la comunicación serial en el PcDuino es necesario habilitar el puerto UART, el primer paso es escribir en la consola de Linux el siguiente código:

$ sudo modprobe gpio

El siguiente paso es configurar el GPIO[27] para que funcione en modo UART, esto se realiza mediante las siguientes líneas de código:

$ echo “3” > /sys/devices/virtual/misc/gpio/mode/gpio0

$ echo “3” > /sys/devices/virtual/misc/gpio/mode/gpio1

Al configurar los puertos *gpio0* y *gpio1* son habilitados para el envío y recepción de datos.

En el PcDuino 3 los pines utilizados para la comunicación serial se muestran en la figura 2.8:

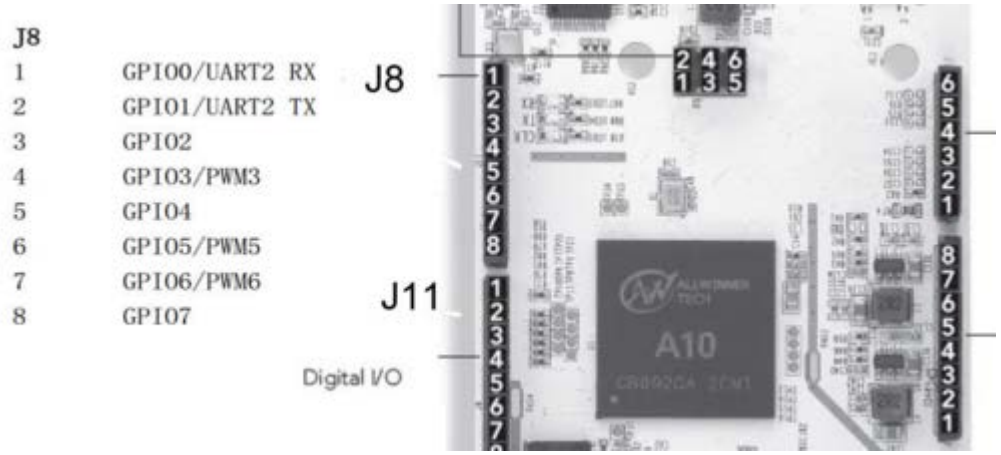


Figura 2.8: Pines PcDuino.
Fuente: (LinkSprite, 2016).

El *pin* número 1 GPIO0/UART2 es el *pin* de recepción, mientras que el *pin* número 2 GPIO1/UART2 es el *pin* de transmisión.

2.3 Comunicación Modbus en Python

Para realizar una comunicación mediante el protocolo Modbus en Python, se necesitan varias librerías adicionales para Python. Una de ellas es PySerial, la cual se utiliza para tener acceso al puerto serial.

[27] GPIO: *General Purpose Input/Output* (Entrada/Salida de Propósito General)

Para la instalación de PySerial, existen dos opciones:

1. Ingresar a la consola (Windows/Linux) y escribir el siguiente código:
   $ python -m pip install pyserial
2. Descargar el archivo desde las fuentes (http://pypi.python.org/pypi/pyserial o https://github.com/pyserial/pyserial/releases), a continuación descomprimir el archivo, ingresar al directorio y escribir el siguiente código en la consola de Windows o Linux:
   $ python setup.py install

Después de la instalación de PySerial en Python se necesita una librería que permita manejar la comunicación y la configuración mediante el protocolo Modbus, en este caso se utiliza la librería minimalmodbus**.**

La instalación de minimalmodbus puede hacerse de dos formas:

1. Ingresar a la consola (Windows/Linux) y escribir el siguiente código:
   $ pip install minimalmodbus
2. Descargar el archivo desde https://pypi.python.org/pypi/MinimalModbus/, a continuación descomprimir el archivo, ingresar al directorio y escribir el siguiente código en la consola de Windows o Linux:
   $ python setup.py install

2.4 Módulos de radio frecuencia

Los módulos HAC-Smart Series UM96/M1 (ver figura 2.9) están basados en un estándar de tecnología *wireless*. Estos módulos tienen una entrada para puerto serial (DB9) y están protegidos por una carcasa de aluminio. Exteriormente cuenta con un *LED* indicador de estado y otro *LED* indicador de transferencia de datos, también cuenta con un *dip-switch* de 8 interruptores la configuración general. Funciona con alimentación de 5V (TECHONOLOGY, 2008).

Figura 2.9: Radio UM96/M1.

Estos módulos son capaces de proveer tres tipos diferentes de estándar de comunicaciones como son TTL, RS232 y RS485.

2.4.1 Configuración del *switch*

Como se observa en la figura 2.10 los módulos tienen un *dip-switch* de configuración de 8 interruptores. Cuando el interruptor está en la posición "*ON*" significa que está en 0, si el interruptor está en la posición "*OFF*" significa que está en 1.

Figura 2.10: Pines de configuración Radio UM96/M1.

2.4.2 Configuración del canal

Los primeros tres interruptores (1, 2, 3) configuran el canal, se elige desde el canal 0 al canal 7. Para lograr una comunicación entre dos módulos es necesario que los dos estén en el mismo canal. Las posibles combinaciones como se observa en la tabla 2.1 son las siguientes:

Tabla 2.1: Tabla de configuración de canal Radio UM96/M1.

Numero de Canal	**Frecuencia**	**Numero de canal**	**Frecuencia**
SW 321=000(0)	La misma frecuencia que un tipo estándar	SW 321=100(4)	La misma frecuencia que un tipo estándar
SW 321=001(1)	La misma frecuencia que un tipo estándar	SW 321=101(5)	La misma frecuencia que un tipo estándar
SW 321=010(2)	La misma frecuencia que un tipo estándar	SW 321=110(6)	La misma frecuencia que un tipo estándar
SW 321=011(3)	La misma frecuencia que un tipo estándar	SW 321=111(7)	La misma frecuencia que un tipo estándar

Fuente: (TECHONOLOGY, 2008).

2.4.3 Selección del modo de paridad.

Los módulos soportan paridad par y sin paridad. Se puede seleccionar la paridad mediante el quinto interruptor. Cuando el interruptor está en 1 la paridad es PAR, por el contrario si el interruptor está en 0 no hay paridad (TECHONOLOGY, 2008).

2.4.4 Modo configuración y modo de comunicación

Mediante el sexto interruptor se cambia a modo configuración o modo de comunicación. En el modo de configuración el módulo permite la conexión al computador para configurar sus diferentes parámetros, mientras que en el modo comunicación el modulo está listo para transmitir con las configuraciones que estén en ese momento.

Los interruptores restantes no influyen en el funcionamiento del módulo de radio frecuencia, por lo tanto no es relevante su estado (TECHONOLOGY, 2008).

2.4.5 Pines del conector Serial (DB9)

Tabla 2.2: Tabla de configuración de pines de Radio UM96/M1.

Pin No	Descripción	Instrucción	Nivel	Conexión al terminal	Conexión al ordenador
1	De reserva				
2	RxD	RxD a RS-323	RS-232	TxD	Tercer pin del ordenador
3	TxD	TxD a RS-232	RS-232	RxD	Segundo pin del ordenador
4	VCC	Fuente de alimentación	Tipo estandar	DC 5v	
5	GND	Puesta a tierra	0	Tierra	Quinto pin del ordenador
6	TxD	TxD a TTL	TTL	RxD	
7	RxD	RxD a TTL	TTL	TxD	
8	A	A a RS-485	RS-485	A	
9	B	B a RS-485	RS-485	B	

Fuente: (TECHONOLOGY, 2008).

2.5 Modulo Xbee

Figura 2.11: Xbee S2.
Fuente: (sparkfun, 2017).

Son módulos que brindan un medio inalámbrico para la comunicación entre dispositivos, los Xbee utilizan el protocolo de comunicación IEEE 802.15.4. Con estos dispositivos es posible crear redes punto a punto y también multipunto. Los módulos utilizados en este proyecto son los Xbee Series2 como se muestra en la figura 2.11.

2.5.1 Configuración Xbee S2

Para configurar el Xbee se necesita el módulo Xbee Explorer USB (ver figura 2.12), este dispositivo permite la conexión entre el Xbee y la PC por medio del cable USB.

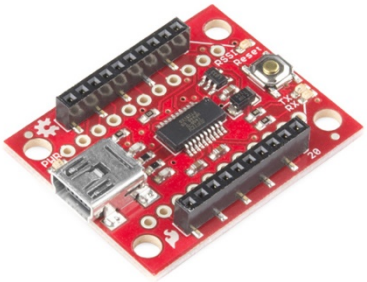


Figura 2.12: Xbee Explorer USB.
Fuente: (sparkfun, 2017)

También es necesaria la instalación del *software* "X-CTU", posterior a la instalación, la configuración del Xbee se realiza mediante los siguientes pasos:

1. Reconocer dispositivo.

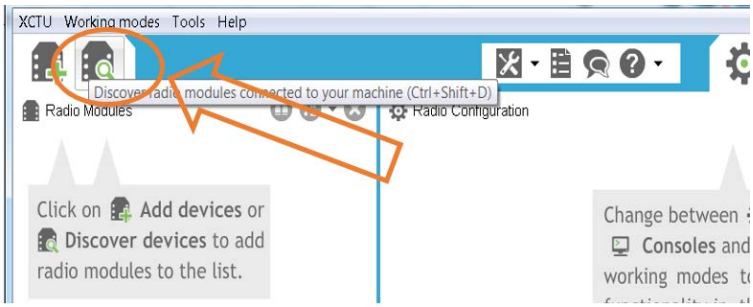


Figura 2.13: *Software* X-CTU

2. Seleccionar el puerto y parámetros de lectura del dispositivo (Velocidad, *bits*, paridad, bits de parada y control de datos).

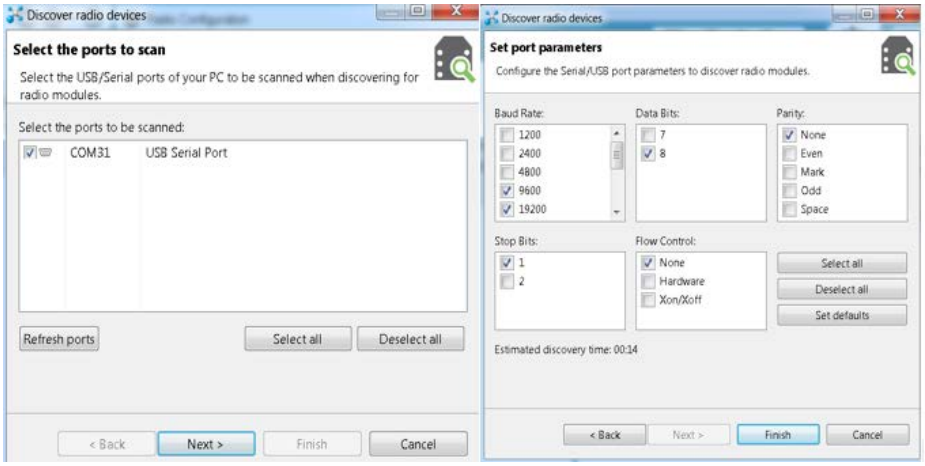


Figura 2.14: Selección puerto y parámetros.

3. El *software* realiza un escaneo y muestra los dispositivos actualmente conectados.

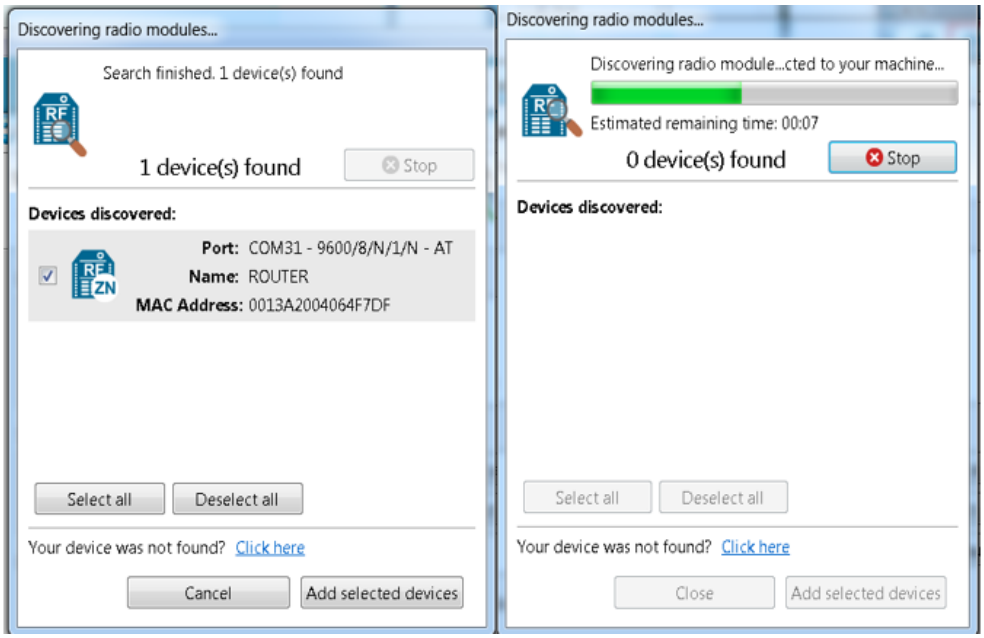


Figura 2. 15: Reconocimiento del módulo, Puerto y dirección MAC.

4. A continuación se muestran todas las propiedades del Xbee se realiza una conexión punto a punto, cada uno de los Xbee tiene su dirección de destino como se muestra en las figuras 2.16 y 2.17, al conectar los dispositivos al PC se identifica las direcciones MAC (figuras 2.18 y 2.19) y para finalizar se configura la interfaz serial del Xbee (figuras 2.20 y 2.21).

Figura 2.16: Xbee A.

Figura 2.17: Xbee B.

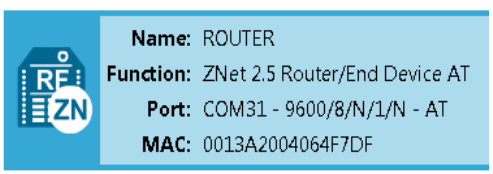


Figura 2.18: Xbee A Características.

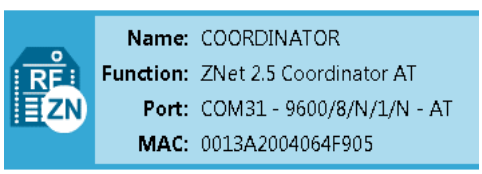


Figura 2.19: Xbee B Características

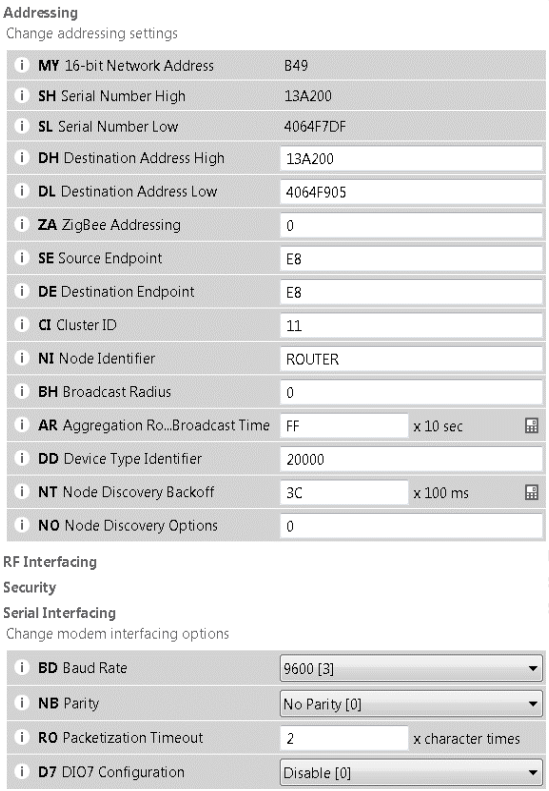


Figura 2.20: Xbee A Configuración.

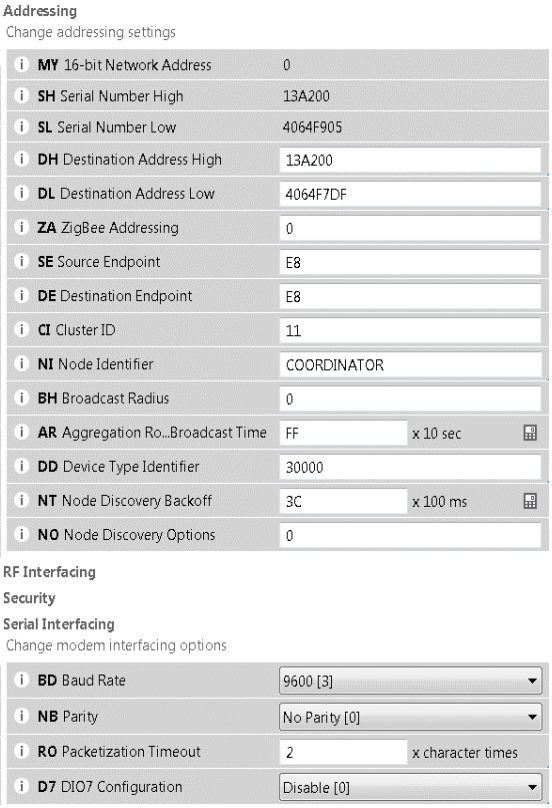


Figura 2.21: Xbee B Configuración.

2.6 Simulación de la comunicación del protocolo ModBus de forma virtual.

2.6.1 Creación del puerto virtual.

Para la creación de un puerto virtual se utiliza el programa "*Virtual Serial Ports Emulator*" (figura 2.22) este permite crear puertos virtuales y simular conexiones entre los puertos creados.

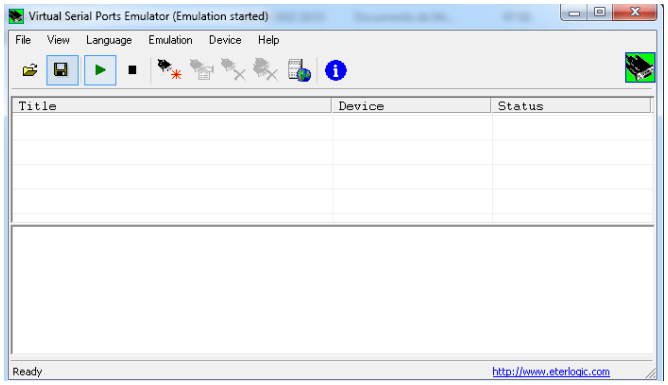


Figura 2.22: Programa "Virtual Serial Ports Emulator".

Para crear un puente virtual entre dos puertos seriales, en el programa seleccionamos la opción "*Pair*", a continuación se elige el número de los puertos, en este caso se seleccionan los puertos COM10 y COM11. Por último se inicia la simulación como se muestra en la figura 2.23.

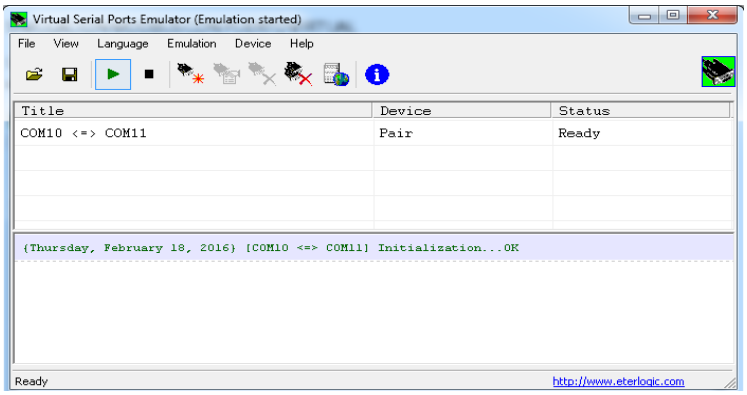


Figura 2.23: Simulación Iniciada.

2.6.2 Introducción a Modbus RTU y Python Spyder.

El programa "Modbus RTU" es un simulador del protocolo Modbus RTU (*Remote Terminal Unit*) ver figura 2.24, este es un *software* de código abierto, en las pruebas es utilizado como esclavo debido a la fácil visualización de datos recibidos.

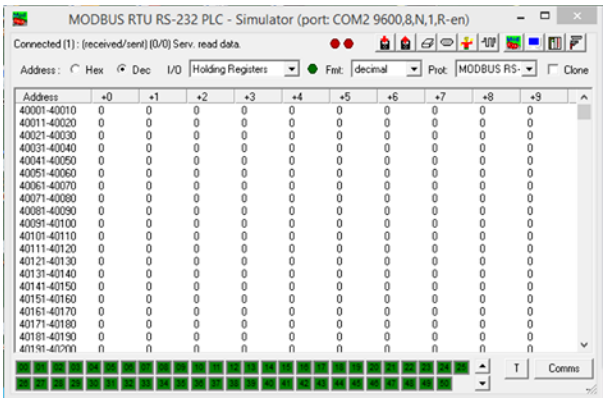


Figura 2.24: Programa "Modbus RTU".

El *software* "Python Spyder" (ver figura 2.25) es un entorno de desarrollo científico en Python basado en Matlab, es utilizado como maestro para enviar datos en Modbus.

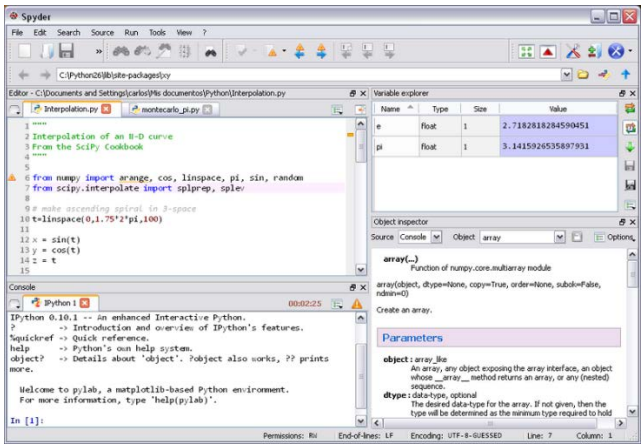


Figura 2.25: Programa "Python Spyder".

2.6.3 Introducción a Modbus Poll y Modbus Slave.

El programa "Modbus Poll" es un simulador del protocolo Modbus y funciona como maestro (ver figura 2.26), puede comandar a varios esclavos, soporta las siguientes funciones principales de Modbus:

01: Leer estado de bobina.

02: Leer estado de entrada.

03: Leer registros de retención.

04: Leer registros de entrada.

05: Fuerza única bobina

06: Preajustar un registro único.

15: Forzar múltiples bobinas.

16: Preajustar múltiples registros.

17: Respuesta de identificación de esclavo.

22: Escritura de registro de mascara.

23: Registro de lectura/escritura.

El programa "Modbus Slave" sirve como esclavo del protocolo ModBus (ver figura 2.26).

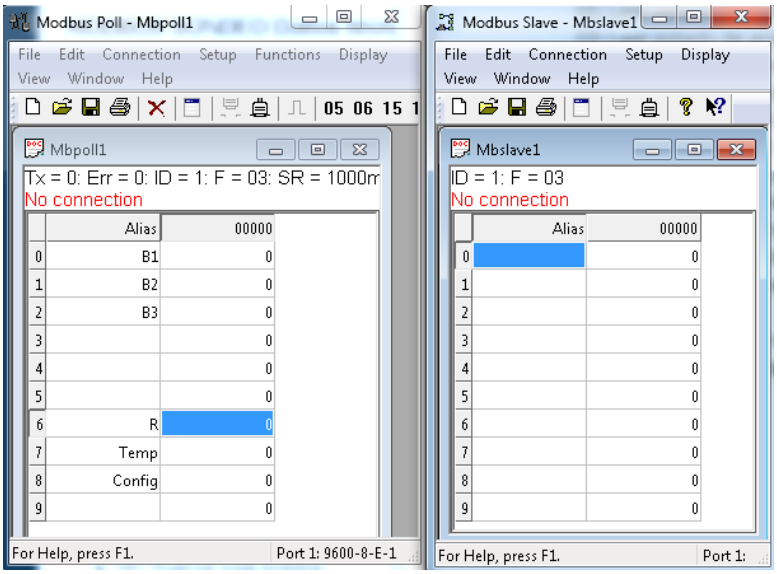


Figura 2. 26: "Modbus Poll" y "Modbus Slave".

2.6.3.1 Configuración maestro esclavo.

Teniendo en cuenta que ya existe el puente virtual entre los puertos COM10 y COM11, el siguiente paso consiste en configurar el maestro y esclavo con la misma velocidad de transmisión, mismo número de *bits* de parada, igual *bits* de datos y misma paridad. El puerto serial que utiliza el maestro es el COM11, mientras que el esclavo utiliza el puerto serial COM10. Todos estos parámetros se muestran en la figura 2.27.

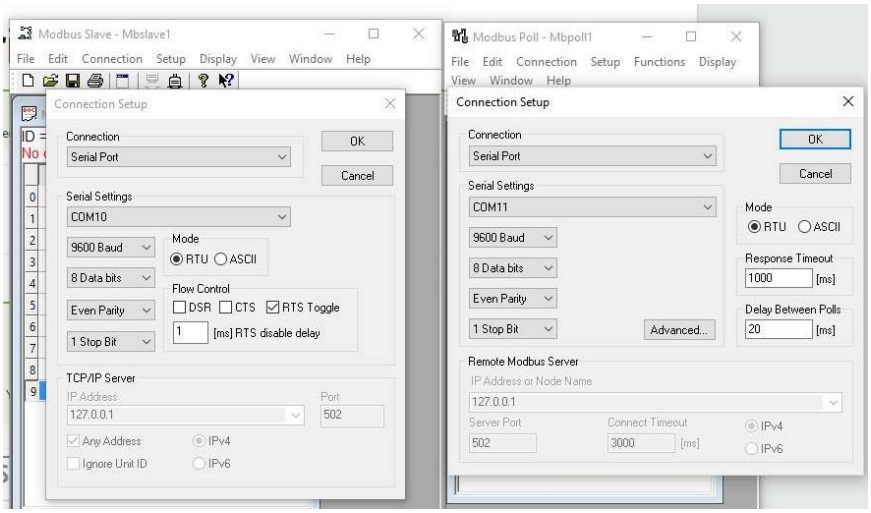


Figura 2.27: Configuraciones de conexión en Modbus Slave, Modbus Poll.

2.6.4 Simulaciones virtuales

2.6.4.1 Modbus Poll a Modbus Slave.

En la figura 2.28 se muestra una conexión entre un maestro y esclavo, a su vez se puede observar que no existe ningún error en la conexión ya que los datos que se escriben en el maestro son los mismos que recibe el esclavo.

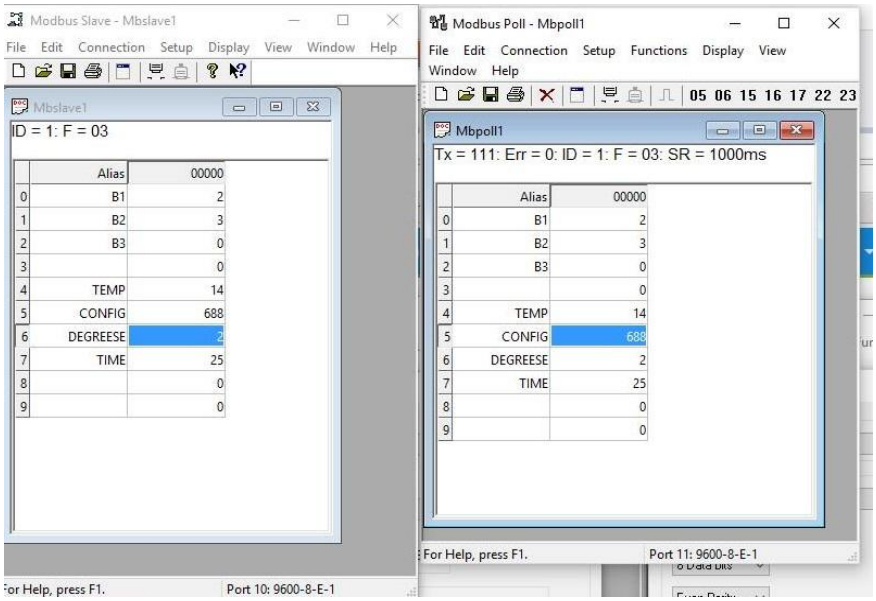


Figura 2.28: Comunicación entre Modbus Poll y Modbus Slave.

2.6.4.2 Modbus Python a Modbus RTU

Para esta simulación se utilizan los programas "Python Spider" para el maestro y para el esclavo "MODBUS RTU", la velocidad de transmisión es de 9600 baudios, no existe *bit* de paridad, los *bits* de datos son 8 y existe un *bit* de parada. El esclavo se conecta a través del puerto COM26, mientras que el maestro se conecta por el puerto COM11. Todos estos parámetros se pueden observar en la figura 2.29.

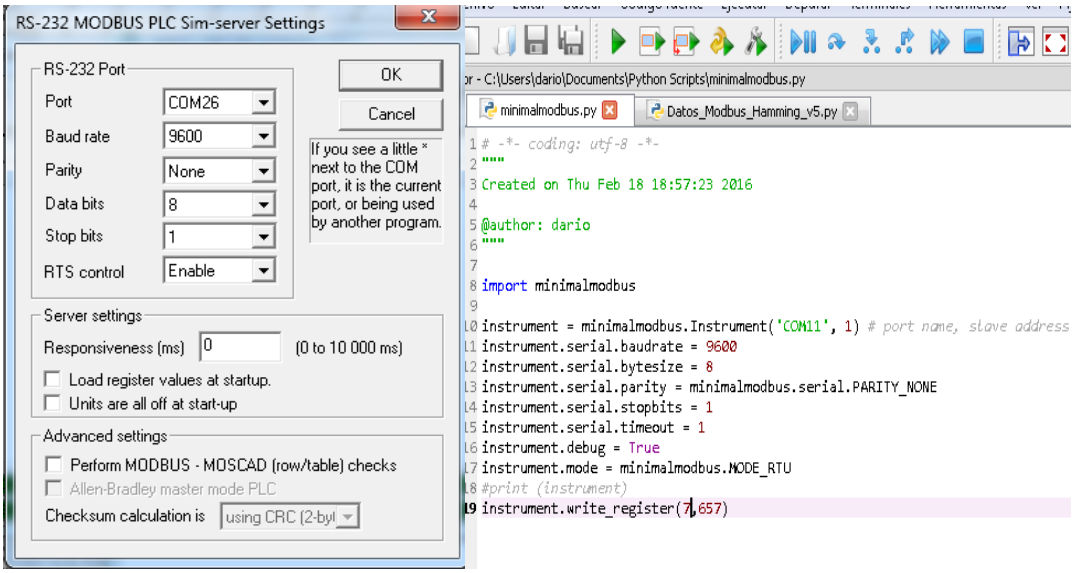


Figura 2.29: Configuraciones de puerto Modbus Python.

Como se observa en la figura 2.29, el código Python hace el papel de maestro utilizando la librería minimalmodbus. Como prueba de conexión el maestro escribe en el registro número 7 el valor 657. En la figura 2.30 se muestra que el esclavo ha recibido el valor enviado desde el maestro, el registro 7 tiene el valor 657.

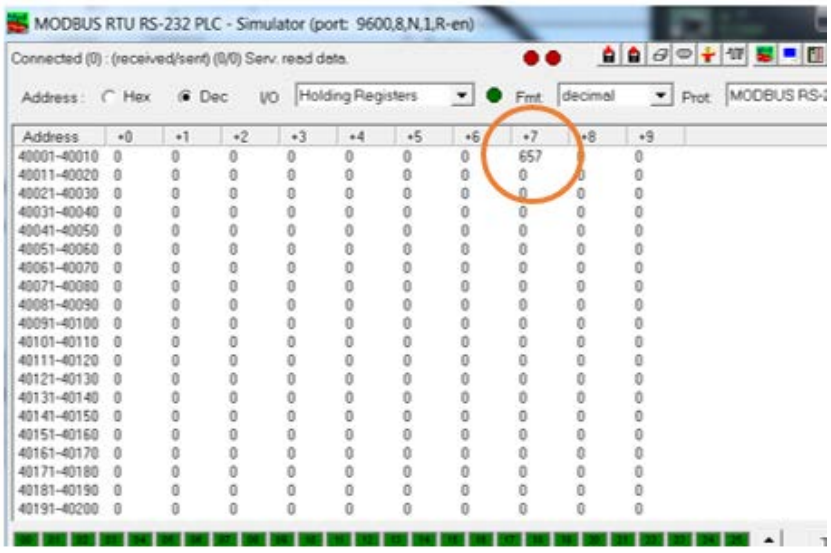


Figura 2.30: Slave Modbus RTU.

2.7 Simulación del protocolo Modbus mediante conexión por cable.

Para este tipo de conexión se utiliza un convertidor de puerto USB a serial RS232, un cable serial punto a punto y un circuito convertidor TTL a RS232.

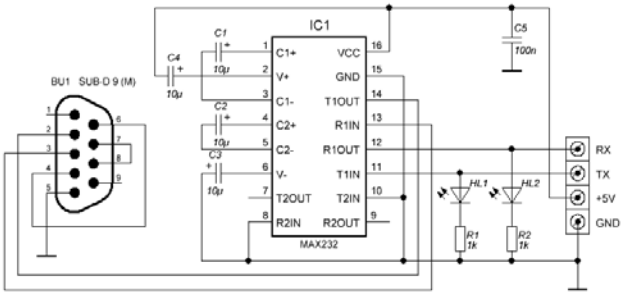


Figura 2.31: Circuito convertidor TTL a RS232
Fuente: (Robomart, 2015).

El circuito permite convertir las señales de transmisión y recepción del puerto serial a señales TTL (ver figura 2.32) para que puedan ser leídas por dispositivos compatibles con TTL como el PcDuino.

2.7.1 Modbus Python a Modbus RTU entre dos PCs.

Para simular una comunicación Modbus desde una PC hasta otra PC, se utiliza dos cables USB a serial y un cable serial normal (ver figuras 2.32 y 2.33). El PC maestro contiene un programa desarrollado en Python que permite modificar un registro.

Figura 2. 32: Conexión cable USB-rs232 PC1.

Figura 2.33: Conexión cable USB-rs232 PC2.

El programa que hace de maestro se observa en la figura 2.34:

```
import minimalmodbus

instrument = minimalmodbus.Instrument('COM9', 1) # port name, slave address (in decimal)
instrument.serial.baudrate = 9600
instrument.serial.bytesize = 8
instrument.serial.parity = minimalmodbus.serial.PARITY_NONE
instrument.serial.stopbits = 1
instrument.serial.timeout = 1

instrument.mode = minimalmodbus.MODE_RTU

instrument.write_register(5,14)
```

Figura 2.34: Programa maestro en Python.

Se utiliza la librería "minimalmodbus" de Python para realizar la comunicación, en la segunda línea del código se define el puerto serial a utilizar en este caso 'COM9' y a continuación la dirección del esclavo. La tercera línea indica la velocidad de transmisión, para este caso es 9600 baudios por segundo. En la cuarta línea se configura el tamaño del *byte* en este caso 8. La quinta línea configura la paridad. En la línea número 6 se configura los *bits* de parada, para este ejemplo se selecciona 1 *bit* de parada. La línea numero 7 configura el *timeout*[28] en este caso específico tenemos 1 ms. Y por último se configura el modo de transmisión en modo RTU.

La última línea sirve para escribir en un registro. El primer valor indica el número de registro y el segundo valor indica el dato que va a contener el registro. En este caso se escribe en el registro numero 5 el valor 14.

[28] *Timeout*: mensaje de error en el maestro cuando el tiempo de espera se agota al conectarse a un esclavo.

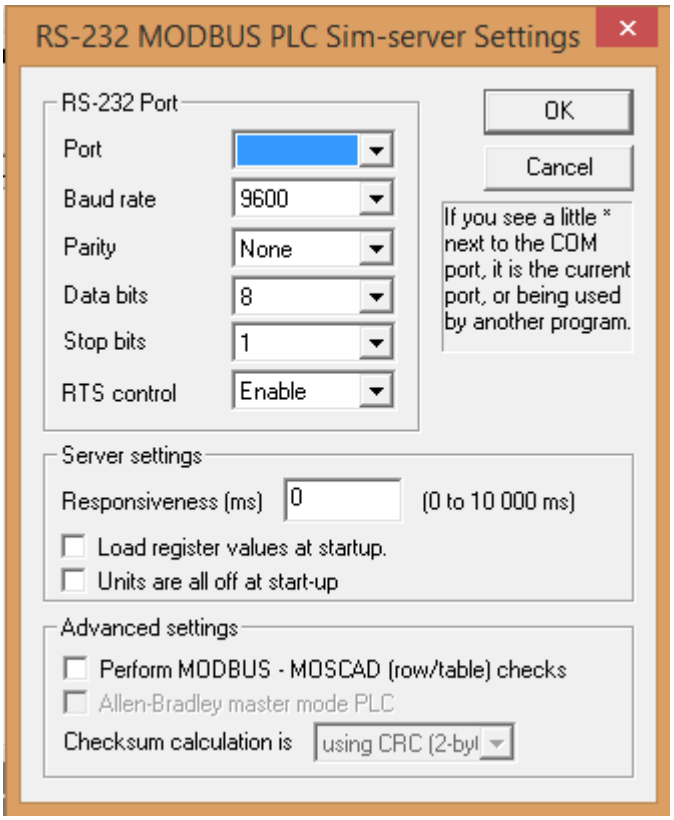


Figura 2.35: Configuración del puerto RS232.

En la figura 2.35 se muestra la configuración del puerto RS232 para el esclavo. Se elige el puerto serial a utilizar y se comprueba que el resto de configuraciones coincidan con las mencionadas anteriormente en el programa desarrollado en Python.

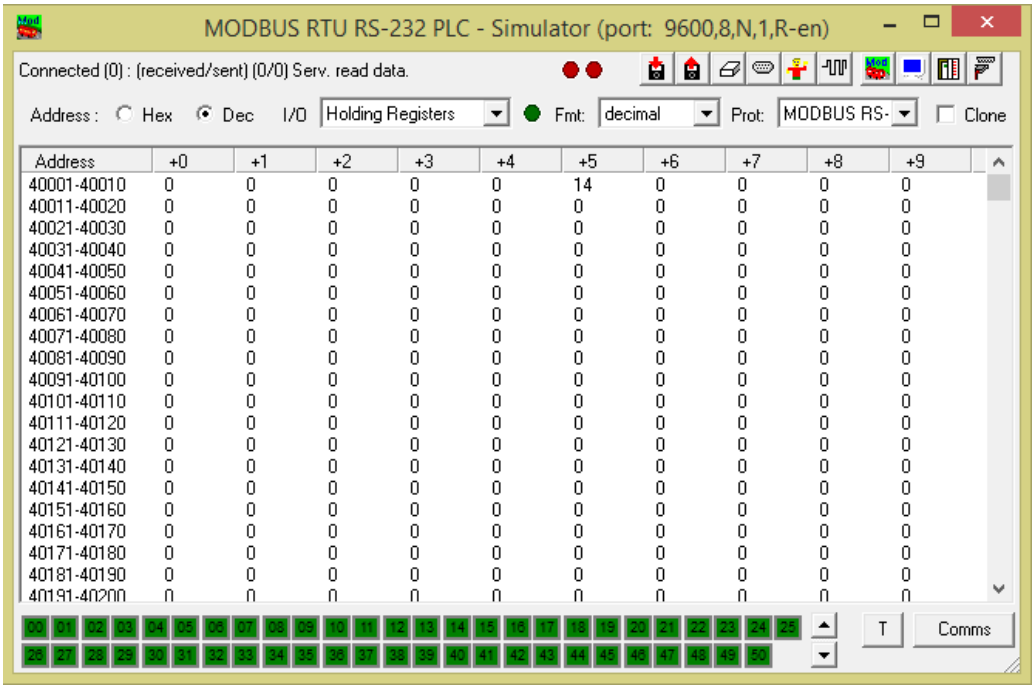


Figura 2.36: Slave Modbus RTU PC1.

En la figura 2.36 se muestran los registros que contiene el esclavo, como no existe error en la conexión, el dato enviado por el maestro (registro 5 valor 14) se recibe correctamente. Con esto se prueba el correcto funcionamiento de la conexión entre esclavo y maestro mediante una conexión cableada.

2.7.2 Modbus Python en el PcDuino a Modbus RTU en el PC.

Para probar la comunicación mediante el protocolo Modbus entre el PcDuino y un PC, el PcDuino se configura como maestro mientras que el PC se configura como esclavo. En el PcDuino se encuentra el código mediante el cual se realiza el envío de datos hacia el maestro. Como se muestra en la figura 2.37 el puerto serial que se utiliza para la comunicación es el "/dev/ttyS1" que es el equivalente a un puerto "COM" en Windows. Los siguientes parámetros de configuración son:

- Velocidad: 9600 baudios.
- Paridad: Ninguna.
- Tamaño del *byte*: 8
- *Bits* de parada: 1
- *Timeout:* 1 ms.
- Escribir en el registro número 9 el valor 9.

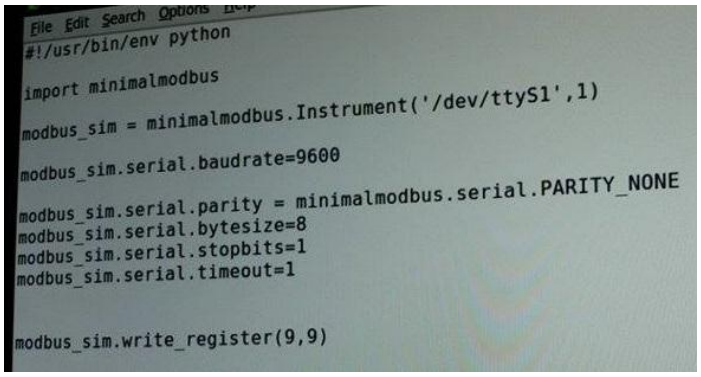


Figura 2.37: Programa PcDuino Pyhton maestro.

La configuración del puerto COM en el esclavo debe contener los mismos valores del maestro para que se establezca la conexión. Como se observa en la figura 2.38 se cumple la orden del maestro, escribir en el registro numero 9 el valor 9. Con esto se prueba el correcto funcionamiento de la conexión entre esclavo y maestro mediante

una conexión cableada con la variante de que el maestro es el PcDuino y el esclavo es el PC.

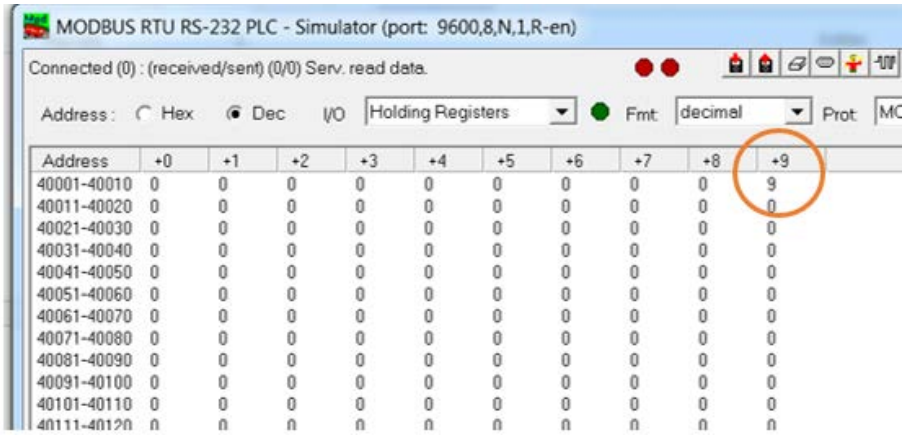


Figura 2.38: Modbus RTU Slave PC.

2.8 Simulaciones mediante conexión inalámbrica.

2.8.1 Introducción.

Las configuraciones para los puertos realizadas anteriormente no varían, en esta prueba se utiliza en lugar de la conexión mediante cable los módulos Xbee y los módulos HAC-Smart Series UM96/M1.

2.8.1.1 Conexión mediante los módulos HAC-Smart Series UM96/M1.

La conexión entre los módulos y los cables USB a serial se muestran en las figuras 2.39 y 2.40, los módulos deben ser alimentados con 5 voltios.

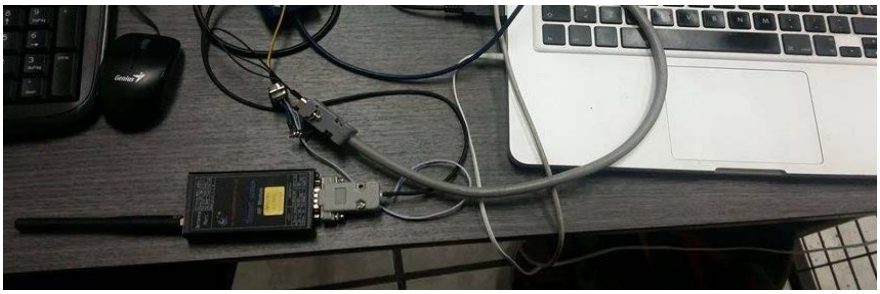


Figura 2.39: Conexión inalámbrica modulo RF PC1.

Figura 2.40: Conexión inalámbrica modulo RF PC2.

Como se observa en las figuras anteriores la conexión entre el cable USB a serial y el modulo RF es directa.

El Programa maestro es el mismo explicado anteriormente y la recepción de datos en el esclavo se muestra a continuación en la figura 2.41:

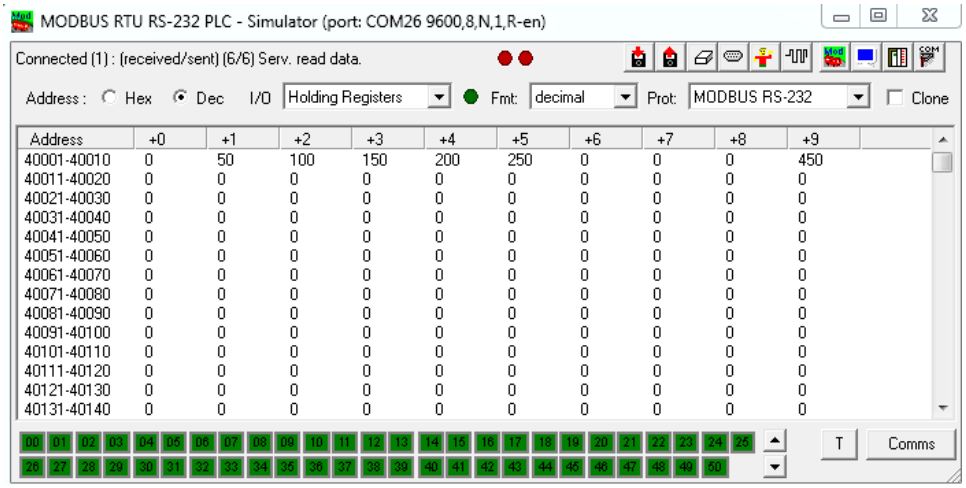


Figura 2.41: Modbus RTU PC1 esclavo.

Los datos enviados son diferentes, pero se comprueba que en efecto el esclavo está recibiendo los datos que el maestro envía como se observa en la figura 2.41.

2.8.1.1 Conexión mediante los módulos Xbee.

En la figura 2.42 se muestra la conexión de los Xbee a las PCs. Para la conexión entre el PC y el Xbee se utiliza un cable USB.

Figura 2.42: Conexión inalámbrica entre PCs mediante módulos Xbee.

En la figura 2.43 se muestran los datos recibidos en el esclavo. El programa maestro es el mismo utilizado anteriormente, para comprobar la comunicación entre el maestro y esclavo, se ejecuta la función de escribir en los registros. En el registro numero 3 (+2) se escribe el valor 10 y en el registro numero 4 (+3) se escribe el valor 5 como se observa en la figura 2.43.

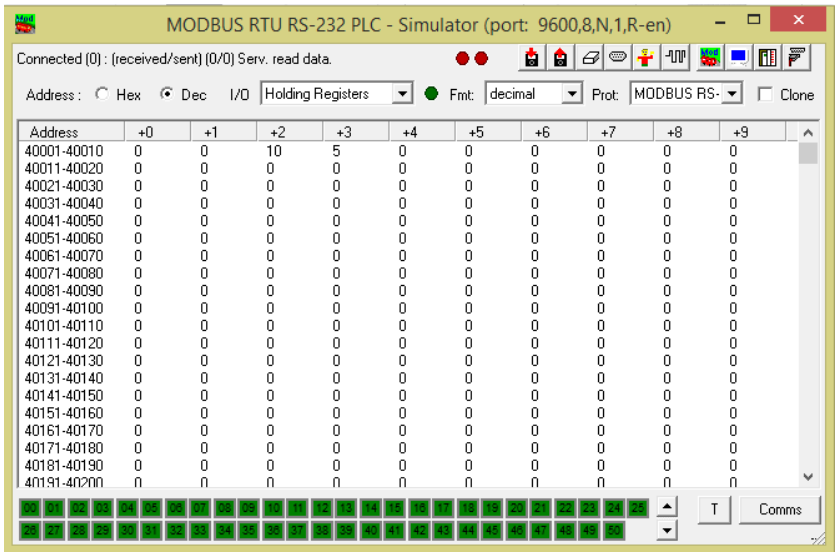


Figura 2.43: Modbus RTU PC2 esclavo.

2.8.2 Modbus Python en el PcDuino a Modbus RTU en el PC.

La conexión del módulo de radiofrecuencia al PcDuino requiere el uso del circuito Max-232 mencionado anteriormente, esto debido a la variación de los niveles de voltaje que entrega el PcDuino en sus pines de transmisión y recepción.

La conexión entre el cable USB a serial es la misma. La conexión entre los módulos de radiofrecuencia y el PcDuino además de la PC pueden observar en la figura 2.44.

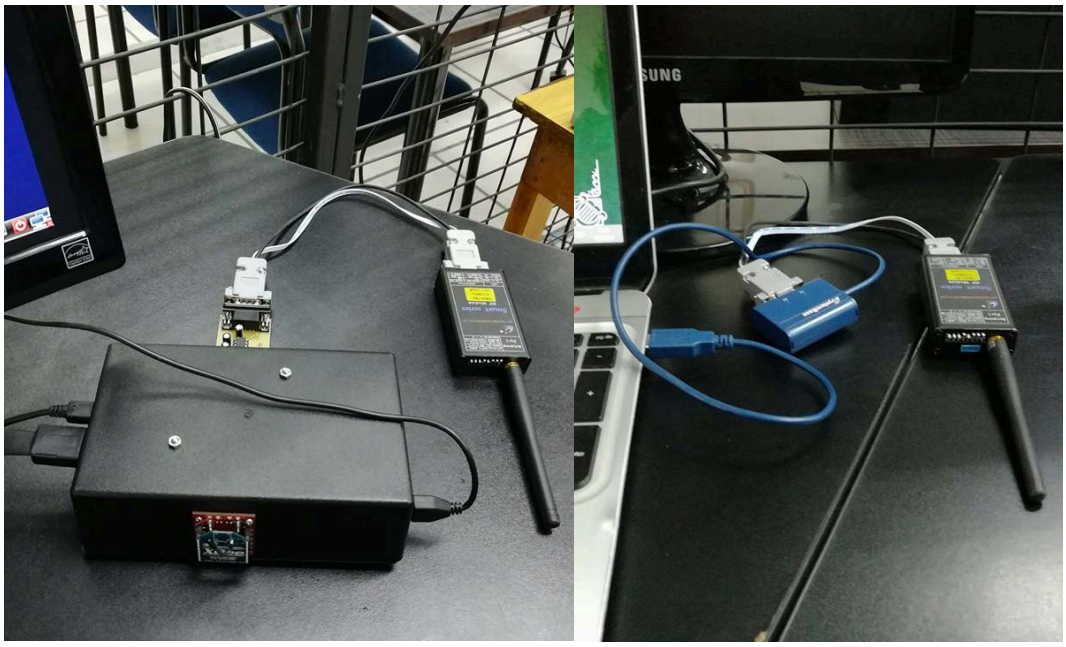


Figura 2.44: Conexión inalámbrica PcDuino y PC.

El PcDuino hace el papel de maestro, la configuración del puerto es la siguiente y se puede observar en la figura 2.45:

- Puerto serial: /dev/ttyS1
- Velocidad: 9600 baudios.
- Paridad: Ninguna.
- Tamaño del *byte*: 8.
- *Bits* de parada: 1
- *Timeout:* 1 ms
- Escribir en el registro número 2 el valor 222.

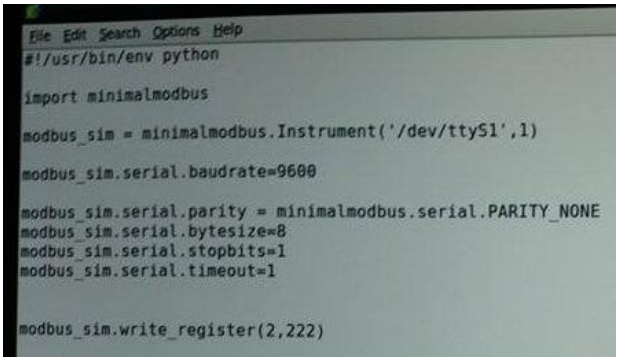


Figura 2.45: Maestro Python en PcDuino conexión inalámbrica.

La configuración del puerto COM en el esclavo debe contener los mismos valores del maestro para que se establezca la conexión. Como se observa en la figura 2.46 se

cumple la orden del maestro, escribir en el registro numero 3 (+2) el valor 222. Con esto se prueba el correcto funcionamiento de la conexión entre esclavo y maestro mediante una conexión inalámbrica entre el PcDuino y el PC.

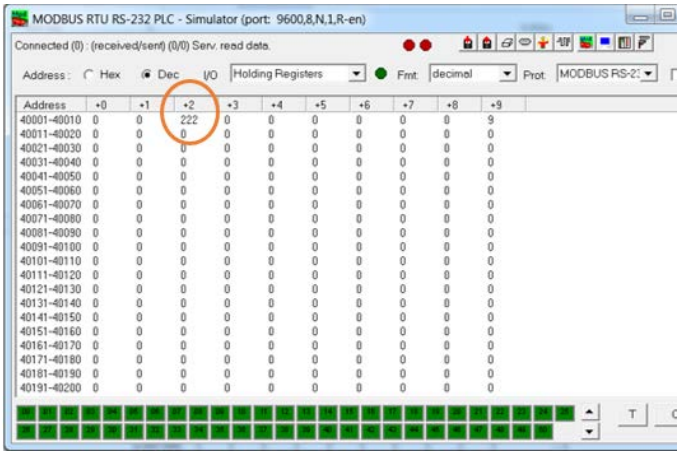


Figura 2.46: Esclavo Modbus RTU PC conexión inalámbrica.

2.9 Conclusiones.

En este capítulo se analiza el *hardware* y *software* utilizado para realizar la comunicación mediante el protocolo Modbus. La implementación del sistema de comunicación se ha realizado mediante dos dispositivos:

- Módulos de radio frecuencia HAC-Smarty
- Módulos Xbee

Para la comunicación mediante el protocolo Modbus se utilizó un sistema embebido basado en el PcDuino.

En este capítulo se realizaron pruebas de funcionamiento del protocolo Modbus entre el maestro y el esclavo con la ayuda de diferentes programas como: Modbus Poll, Modbus Slave.

CAPÍTULO 3

DISEÑO E IMPLEMENTACIÓN DE LA COMUNICACIÓN MODBUS ENTRE EL CODIFICADOR Y DECODIFICADOR HAMMING EN LENGUAJE PYTHON

3.1 Introducción.

En este capítulo se realiza el estudio de las diferentes librerías de Python implementadas en el protocolo de comunicaciones industriales ModBus, mediante la librería Modbus-tk se desarrollan programas de maestro y esclavo que sirven para realizar la comunicación del sistema. Además se explican las diferentes funciones y códigos de excepción del protocolo ModBus, implementando las técnicas de corrección de errores como el código Hamming.

Por último se realiza el desarrollo de los programas codificadores y decodificadores mediante el código Hamming que se implementan en el maestro y esclavo para él envió y recepción de datos.

3.2 Tipos de librerías Modbus en Python.

3.2.1 Pymodbus.

Pymodbus es una implementación del protocolo Modbus para Python. Pymodbus se puede utilizar sin la necesidad de otras librerías adicionales, salvo la librería PySerial. Funciona en las versiones de Python desde la 2.3 hasta la 3.0.

3.2.1.1 Características.

3.2.1.1.1 Características del Cliente.

- Protocolo completo de lectura/escritura en el registro.
- TCP, UDP, Serial ASCII, Serial RTU y Serial Binario.
- Versiones síncronas y asíncronas.

3.2.1.1.2 Características del Servidor

- Puede funcionar como un servidor ModBus.
- TCP, UDP, Serial ASCII, Serial RTU y Serial Binario.
- Versiones síncronas y asíncronas.
- Control completo del servidor (información del dispositivo, contadores, etc.)
- Una serie de contextos de soporte: base de datos, redes, motor de base de datos en memoria, dispositivo esclavo. (GitHub, minimalmodbus, 2017).

3.2.2 MinimalModbus 0.7

3.2.2.1 Características.

MinimalModbus es un módulo de Python presenta facilidades para la comunicación con instrumentos (esclavos) desde un computador (maestro) utilizando el protocolo Modbus, y está destinada a estar en ejecución en el maestro. La única dependencia es el módulo PySerial.

Este *software* soporta las versiones de comunicación serial Modbus RTU y Modbus ASCII, está diseñado para su uso en Linux, OS X y para Windows. Es de código abierto. Funciona en las versiones de Python 2.7, 3.2, 3.3 y 3.4 (Berg, 2016).

3.2.3 Modbus-tk.

3.2.3.1 Características.

- Soporta Modbus TCP para escritura en maestro y esclavo
- Soporta Modbus RTU para la escritura de maestros y esclavos (se necesita PySerial)
- Se puede personalizar con mecanismos de enganche (simular errores, tiempos de espera, etc.)
- Define muy fácilmente sus bloques de memoria.
- Coloca y busca valores en cualquier lugar de un bloque de memoria.
- Tiene la capacidad de identificarse a través de diferentes módulos de registro Python. (GitHub, modbustk, 2017).

3.3 Implementación de Modbus-tk en Python para la comunicación Maestro Esclavo.

Modbus Test Kit o Modbus-tk es una implementación del protocolo Modbus en Python que hace posible trabajar como Modbus TCP y Modbus RTU, puede ser utilizado para pruebas tanto de maestro como de esclavo. Se puede utilizar para crear cualquier aplicación que necesite comunicarse a través de Modbus y se utiliza en aplicaciones "reales". Gracias a Python y el extenso conjunto de bibliotecas existentes, puede adaptarse a una gran cantidad de necesidades como: registro de base datos, HMI[29], generación de informes.

Para tener una idea sobre el funcionamiento del maestro y del esclavo, a continuación se presentan los flujogramas de cada uno.

3.3.1 Maestro.

Inicialmente se configuran los parámetros del puerto de comunicación (puerto COM, número de esclavo, velocidad de transmisión, paridad, *bits* de parada.). Finalizada la configuración del puerto, el maestro envía una orden u órdenes al esclavo. Si no existen errores en la comunicación la orden es recibida por el esclavo y el maestro espera la respuesta, pero si existe un error de comunicación el esclavo envía un código de excepción al maestro.

[29] HMI: *Human Machine Interface* (Interfaz hombre máquina)

3.3.2 Flujograma (Maestro).

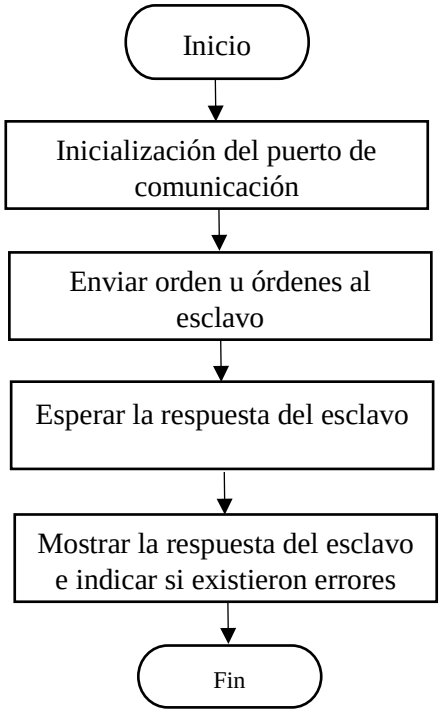


Figura 3.1: Diagrama de Flujo del funcionamiento del Maestro.

3.3.3 Esclavo

Como primer paso se realiza la configuración del puerto de comunicación (puerto COM, velocidad de transmisión, paridad, etc.), luego el esclavo entra en un modo de espera hasta que el maestro le envíe una orden. Si no existe ningún problema en la comunicación el esclavo recibe la orden, envía la respuesta al maestro notificando si la orden fue ejecutada o si existió algún tipo de error en la ejecución (por ejemplo: no existe el número de registro en el cual se iba a escribir cierto valor).

3.3.4 Flujograma Esclavo

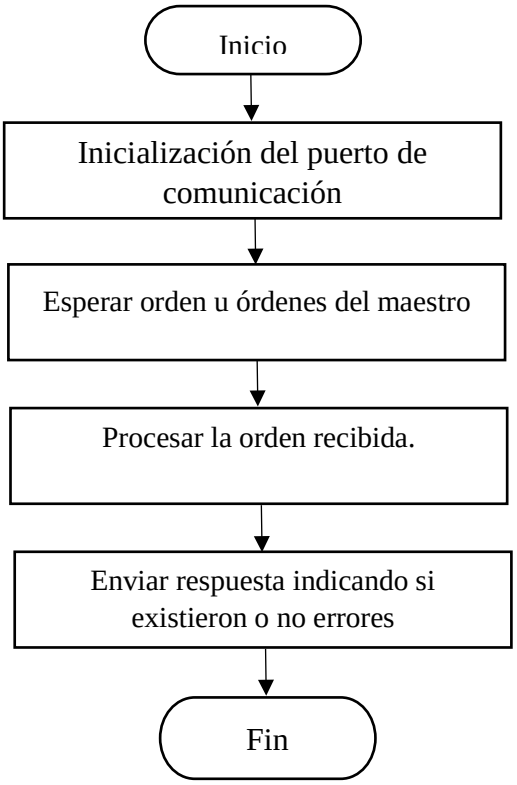


Figura 3.2: Diagrama de flujo del funcionamiento del Esclavo.

3.4 Principales funciones del protocolo Modbus

Para la comunicación entre el maestro y esclavo, la librería Modbus-tk ofrece diferentes funciones propias del protocolo Modbus que sirven para intercambiar información. A continuación se enumeran las funciones que dispone Modbus-tk, así como también las excepciones que se generan y por último los tipos de bloques con que cuenta Modbus-tk.

3.4.1 Funciones Modbus soportadas.

READ_COILS = 1 (Leer Bobinas)
READ_DISCRETE_INPUTS = 2 (Leer entradas discretas/binarias)
READ_HOLDING_REGISTERS = 3 (Leer los registros de retención)
READ_INPUT_REGISTERS = 4 (Leer los registros de entrada)
WRITE_SINGLE_COIL = 5 (Modificar el estado de una sola bobina)
WRITE_SINGLE_REGISTER = 6 (Escribir en un solo registro)
READ_EXCEPTION_STATUS = 7 (Leer el estado de las excepciones)
DIAGNOSTIC = 8 (Diagnóstico)

WRITE_MULTIPLE_COILS = 15 (Escribir en múltiple bobinas)
WRITE_MULTIPLE_REGISTERS = 16 (Escribir en múltiples registros)
READ_WRITE_MULTIPLE_REGISTERS = 23 (Leer/Escribir en múltiples registros).

La mayoría de estas funciones y sus características están descritas anteriormente en el capítulo 1. Para la implementación de las funciones con las que cuenta Modbus-tk, es importante desarrollar los códigos de acuerdo a la estructura de cada código.
A continuación se muestran dichas funciones y su código para Python.

3.4.1.1 Leer bobinas

master.execute(1, cst.READ_COILS, 0, 10)
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.READ_COILS"* indica la función, en este caso leer bobinas.
El tercer dato *"0"*, indica desde que bobina empezará la lectura.
El cuarto dato *"10"*, indica hasta que bobina se realizará la lectura.

3.4.1.2 Leer entradas binarias.

master.execute(1, cst.READ_DISCRETE_INPUTS, 0, 8)
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.READ_DISCRETE_INPUTS"* indica la función, en este caso leer entradas binarias.
El tercer dato *"0"*, indica desde que entrada empezará la lectura.
El cuarto dato *"8"*, indica hasta que entrada se realizará la lectura.

3.4.1.3 Leer registros de entrada.

master.execute(1, cst.READ_INPUT_REGISTERS, 100, 3)
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.READ_INPUT_REGISTERS"* indica la función, en este caso leer los registros de entrada.
El tercer dato *"100"*, indica el número de datos a leer en el registro.
El cuarto dato *"3"*, indica el número de registro en el que se realizará la lectura.

3.4.1.4 Leer registros de retención.

master.execute(1, cst.READ_HOLDING_REGISTERS, 100, 12)
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.READ_HOLDING_REGISTERS"* indica la función, en este caso leer los registros de retención.
El tercer dato *"100"*, indica el número de datos a leer en el registro.
El cuarto dato *"12"*, indica el número de registro en el que se realizará la lectura.

3.4.1.5 Modificar el estado de una sola bobina.

master.execute(1, cst.WRITE_SINGLE_COIL, 7, output_value=1)
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.WRITE_SINGLE_COIL"* indica la función, en este caso modificar el estado de una sola bobina.
El tercer dato *"7"*, indica el número de bobina que será modificada.
El cuarto dato *"output_value=1"*, indica el nuevo valor de la bobina, puede ser *1* o *0*.

3.4.1.6 Modificar el valor de un registro.

master.execute(1, cst.WRITE_SINGLE_REGISTER, 100, output_value=54)
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.WRITE_SINGLE_REGISTER"* indica la función, en este caso modificar el valor de un registro.
El tercer dato *"100"*, indica el número de registro que será modificado.
El cuarto dato *"output_value=54"*, indica el nuevo valor del registro.

3.4.1.7 Modificar el estado de varias bobinas.

master.execute(1, cst.WRITE_MULTIPLE_COILS, 0, output_value=[1, 1, 0, 1, 1, 0, 1, 1])
El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.
El segundo dato *"cst.WRITE_MULTIPLE_COILS"* indica la función, en este caso modificar el estado de varias bobinas.
El tercer dato *"0"*, indica el número de la bobina desde la cual se empezará a cambiar el estado.

El cuarto dato *"output_value=[1, 1, 0, 1, 1, 0, 1, 1]"*, indica los nuevos valores de los estados de las bobinas.

3.4.1.8 Modificar los valores de múltiples registros.

master.execute(1, cst.WRITE_MULTIPLE_REGISTERS, 100,
output_value=xrange(12))

El primer dato *"1"* indica el número del esclavo al cual va dirigido la orden.

El segundo dato *"cst.WRITE_MULTIPLE_REGISTERS"* indica la función, en este caso modificar los valores de múltiples registros.

El tercer dato *"100"*, indica el número de registro desde el cual se empezará a modificar.

El cuarto dato *"output_value= xrange(12)"*, indica los nuevos valores de los registros modificados.

3.4.2 Códigos de Excepciones Generadas

ILLEGAL_FUNCTION = 1 (Función no permitida)
ILLEGAL_DATA_ADDRESS = 2 (Dirección de datos no permitido)
ILLEGAL_DATA_VALUE = 3 (Datos erróneos)
SLAVE_DEVICE_FAILURE = 4 (Falla en el dispositivo esclavo)
COMMAND_ACKNOWLEDGE = 5 (Reconocimiento de comando)
SLAVE_DEVICE_BUSY = 6 (Esclavo ocupado)
MEMORY_PARITY_ERROR = 8 (Error en la paridad de memoria)

3.4.3 Tipos de bloques soportados.

COILS = 1 (Bobinas)
DISCRETE_INPUTS = 2 (Entradas discretas/binarias)
HOLDING_REGISTERS = 3 (Registros de retención) (GitHub, modbustk, 2017).

3.5 Técnicas de detección y corrección de errores.

La detección de errores en los sistemas digitales consiste en verificar únicamente dos posibilidades 1 o 0, por esta razón la corrección se vuelve sencilla ya que se invierten los valores de los *bits* en caso de existir el error. El primer paso para la corrección consiste en determinar la cantidad de *bits* erróneos y la ubicación dentro de la trama de *bits*.

Para la corrección de errores se pueden utilizar dos formas:

3.5.1 El *Backwards Error Correction (BEC)*

La técnica BEC (Corrección de errores hacia atrás), es una técnica de corrección de error en la cual el dispositivo receptor envía una solicitud al dispositivo fuente para que reenvíe la información. Esta técnica se utiliza cuando los datos transmitidos se han perdido o dañado durante la comunicación y el dispositivo de transmisión debe volver a reenviar la información con el fin de que el dispositivo reciba correctamente la información. (Torres, 2014)

3.5.2 *Foward Error Correction (FEC)*

La técnica FEC (Corrección de errores hacía adelante), es una técnica de corrección de errores que tiene como objetivo reconstruir la información deteriorada por los errores, la reconstrucción se realiza en el equipo receptor, para esto se deben utilizar en los códigos un gran número de *bits* de chequeo adicional lo que disminuye la efectividad del código. (Torres, 2014)

Esta técnica implica la codificación de un mensaje de manera redundante, lo que permite al receptor reconstruir los *bits* perdidos sin la necesidad de retransmitirlos.

El *"Forward Error Correction"* es lo contrario de *"Backwards Error Correction"*, en el que un dispositivo de transmisión envía la información más los bits de redundancia para subsanar posibles errores.

Los sistemas FEC se utilizan en el control de paridad a través de paquetes de red y se lleva a cabo por medio de la adición de redundancia a la información transmitida utilizando algoritmos predeterminados como: Hamming, CRC[30], Reed Solomon, BCH[31], Convolucionales, para determinar los errores y corregirlos.

A continuación se muestran algunas técnicas de corrección de errores hacia adelante:

- **Código Hamming**

  El código Hamming se define como un código binario, el mismo que es utilizado para detectar y corregir un solo error en un palabra de n *bits*.

[30] CRC: *Cylic Redundancy Check* (Comprobación de Redundancia Cíclica)
[31] BCH: *Bose, Chaudhuri, Hocquenghem.*

- **Códigos Bose Chaudhuri y Hocquenghem (BCH)**
  Los códigos BCH forman una de las clases más grandes de códigos cíclicos de corrección de errores. Esta clase de códigos son una generalización considerable de los códigos Hamming para la corrección de múltiples errores. Los códigos binarios BCH fueron inventados por *Hocquenghem* en 1959 y de forma independiente por *Bose* y *Chaudhuri* en 1960 (Torres, 2014).

- **Códigos Reed-Solomon.**
  Los códigos Reed-Solomon (códigos RS) son una subclase de los códigos BCH no binarios, ya que el codificador de un código RS opera sobre un bloque de *bits* en vez de *bits* individuales como es el caso de los códigos binarios. De esta manera, los datos son procesados en porciones de m *bits*, llamados símbolos (Torres, 2014).

- **Códigos convolucionales.**
  Los códigos convolucionales son usados extensamente en los sistemas de comunicación inalámbricos y generan *bits* redundantes para el reconocimiento y control de errores de una forma continua, tienen la ventaja de ser códigos con memoria. (Torres, 2014).

Después de analizar las diferentes técnicas de corrección de errores, se puede decir que la técnica más apropiada para este proyecto es el código Hamming por su sencillez de aplicación con respecto a otras técnicas y por la necesidad de corregir únicamente un *bit* en los datos que se intercambian entre maestro y esclavo.
A continuación se detalla más a profundidad el código Hamming y como se realiza la codificación y decodificación con el mismo.

3.6 Fundamentos teóricos del código Hamming.

3.6.1 Plano Proyectivo y Código de Hamming.

El código de Hamming se puede obtener a partir de un plano proyectivo[32] de la figura 3.3.

[32] Un plano proyectivo consiste en un conjunto de líneas, un conjunto de puntos, y una relación entre los puntos y las líneas de incidencia. El plano proyectivo finito más pequeño está definido sobre el

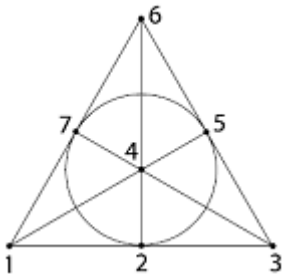


Figura 3.3: Plano de Fano.

Para obtener el código de Hamming se parte del plano de Fano que está compuesto por 7 puntos y 7 líneas (6 rectas y la circunferencia), a los cuales se asocia un valor binario que corresponde a cada línea. (Torres, 2014)

Para encontrar el valor binario se le asigna el valor de 1 a los puntos que forma la línea y el resto de puntos se le asigna el valor de 0, o inversamente, como se puede observar en la tabla 3.1.

Tabla 3.1: Valor binario de acuerdo al plano proyectivo.

Línea	**Valor binario**		**Línea**	**Valor binario**	
V{1 ,2 ,3}	1110000	0001111	V{2 ,4 ,6}	0101010	1010101
V{3 ,5 ,6}	0010110	1101001	V{3 ,4 ,7}	0011001	1100110
V{1 ,6 ,7}	1000011	0111100	V{2 ,5 ,7}	0100101	1011010
V{1 ,4 ,5}	1001100	0110011			

Fuente: (Torres, 2014).

Del plano proyectivo se obtiene los diferentes desplazamientos cíclicos que corresponden a aquellos puntos de dos rectas que se intersectan como {1, 2, 7}, {2, 3, 5}, {5, 6, 7}.

Conforme al plano proyectivo y ordenando la tabla 3.1, se obtienen los códigos Hamming para un desplazamiento cíclico {5, 6, 7} como se observa en la tabla 3.2

cuerpo de dos elementos, y tiene 7 puntos y 7 rectas. Este plano recibe el nombre de plano de Fano. (Torres, 2014)

Tabla 3.2: Códigos de Hamming.

Número	Mensaje de Texto	Mensaje con Código de Hamming	Número	Mensaje de Texto	Mensaje con Código de Hamming
0	0000	0000000	8	1000	1000011
1	0001	0001111	9	1001	1001100
2	0010	0010110	10	1010	1010101
3	0011	0011001	11	1011	1011010
4	0100	0100101	12	1100	1100110
5	0101	0101010	13	1101	1101001
6	0110	0110011	14	1110	1110000
7	0111	0111100	15	1111	1111111

Fuente: (Torres, 2014).

3.6.2 Código de Hamming (7, 4).

El valor (7, 4) representa a un mensaje que tiene cuatro *bits* de datos (D) y va a ser transmitido como una palabra de código de 7 *bits* con la adición de tres *bits* de control de error. A esto se le conoce como código Hamming (7, 4). Los tres *bits* que se agregan son *bits* de paridad (P), donde se calcula la paridad de cada uno de los diferentes subconjuntos de los *bits* del mensaje.

Conforme al plano de Fano el patrón de desplazamiento cíclico lineal {5, 6, 7} está relacionado con los *bits* de datos (1, 2, 3, 4) y se puede representar por medio de una intersección de tres conjuntos como se indica en la figura 3.4. (Torres, 2014)

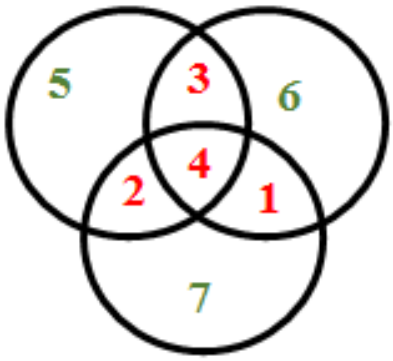


Figura 3.4: Patrón de desplazamiento cíclico {5, 6, 7}. Hacer Gráfico

En la tabla 3.3 se muestra el procedimiento de cómo se obtienen los bits de paridad para el desplazamiento cíclico {5, 6, 7}.

Tabla 3.3: Código de Hamming (7,4).

1	2	3	4	5	6	7	
D	D	D	D	P	P	P	Palabra de código de 7 *bits*
-	D	D	D	P	-	-	Paridad Par
D	-	D	D	-	P	-	Paridad Par
D	D	-	D	-	-	P	Paridad Par

Se observa que el cambio del cuarto *bit* afecta a los tres *bits* de paridad, en cualquier otro caso solo afecta a dos *bits* de paridad.

De acuerdo a la tabla 3.4, el mensaje 1101 será enviado como 11001100 que es el mismo valor que corresponde al obtenido directamente por el plano de Fano.

Tabla 3.4: Análisis de los bits de paridad.

1	2	3	4	5	6	7	
1	1	0	1	0	0	1	Palabra de código de 7 bits
-	1	0	1	0	-	-	Paridad Par
1	-	0	1	-	0	-	Paridad Par
1	1	-	1	-	-	1	Paridad Par

En la tabla anterior se observa que si se produce un error en cualquiera de los *bits* de datos, el error afecta a diferentes combinaciones de los tres *bits* de paridad, dependiendo de la posición del *bit.*

3.7 Codificación y decodificación Hamming utilizando matrices.

Este estudio pretende mostrar un algoritmo de codificación y decodificación usando síndromes que se forman a partir del plano de Fano.

3.7.1 Codificación.

En este análisis se utiliza un código Hamming (7, 4), esto quiere decir que para codificar 4 *bits* de datos se le agrega 3 *bits* de paridad. El código Hamming (7, 4) puede corregir y detectar errores de un solo *bit*, además puede detectar errores de dos *bits* pero no corregirlos.

El vector de código y puede obtenerse por una multiplicación de matrices:

$$y = xG \qquad \text{(EC 3.1)}$$

$x =$ *Bits* de la palabra del mensaje

$G =$ Matriz generadora con la siguiente estructura:

$$G = [I_k|P] \quad \text{(EC 3.2)}$$

dónde:

$I_k =$ Matriz identidad de $k \cdot k$

$k =$ Número de *bits* de datos

$P =$ Matriz de paridad.

$$P = \begin{bmatrix} P_{11} & P_{12} & \cdots & P_{1q} \\ P_{21} & P_{22} & \cdots & P_{2q} \\ \vdots & \vdots & \vdots & \vdots \\ P_{k1} & P_{k2} & \cdots & P_{kq} \end{bmatrix} \quad \text{(EC 3.3)}$$

La fórmula (EC 3.2) indica la matriz generadora, que se utiliza en la implementación del codificador Hamming, para lo cual se utilizará una matriz identidad I_k de $4 \cdot 4$ y como matriz de paridad P de $4 \cdot 3$. La matriz P está formada por los últimos 3 bits de acuerdo a los valores obtenidos del plano proyectivo o plano de Fano. (Torres, 2014)

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Una palabra de mensaje denominada (x) de 4 *bits* se considera como un vector de longitud 4, o equivale a una matriz de $1 \cdot 4$. La palabra codificada que corresponde a (x) es $x \cdot G$, que es una matriz de $1 \cdot 7$, o simplemente una cadena de *bits* de longitud 7. Si el mensaje es $x = 1101$ se codifica de la siguiente manera:

$$x \cdot G = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Las operaciones realizadas anteriormente brindan una idea de cómo se forma la estructura del código Hamming, utilizando algebra lineal.

3.7.2 Decodificación

La matriz de comprobación de paridad H se obtiene a partir de la siguiente estructura:

$$H = [P^T | I_r] \quad \text{(EC 3.4)}$$

dónde:

$P^T =$ Matriz de paridad transpuesta

$I_r =$ Matriz identidad de $r \cdot r$

$r =$ Número de *bits* de cheque o redundancia

La matriz de chequeo a partir de la fórmula (EC 3.4) es la siguiente:

$$H = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Cuando se trabaja con un desplazamiento cíclico {5, 6, 7}, permite a la matriz de comprobación de paridad H ordenarse de la siguiente manera:

$$H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Está matriz H de forma ordenada se utiliza para el análisis y la implementación del decodificador Hamming, el cual permite analizar de manera más sencilla que *bit* es el que contiene error. (Torres, 2014)

Para el análisis de la decodificación se parte del mensaje codificado con código Hamming $y = 1101001$:

$$y = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

El mensaje codificado y se ha escrito como un vector columna, es decir como una matriz de $7 \cdot 1$, en lugar de un vector fila. El producto de la matriz es el siguiente:

$$H \cdot y = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

El resultado de la multiplicación de las dos matrices es el vector cero, esto indica que y es una palabra de código válido, por lo que se toma sus primeros cuatro caracteres 1101 que representa la palabra del mensaje original (Torres, 2014). Suponiendo que el mensaje "y" ha sufrido un error en un *bit* durante la transmisión, el *bit* alterado es el tercero, por lo que en lugar de recibir la palabra y, se recibe la palabra y'.

$$y' = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

El *bit* que contiene el error se obtiene calculando el producto de la matriz:

$$H \cdot y' = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}$$

Como el resultado de la multiplicación no es el vector cero, esto indica que y' no es una palabra de código válido. Si se supone que existe un *bit* con error que se produjo durante la transmisión, para determinar cuál de los siete *bits* de y' es el que contiene el error, se procede a analizar el resultado de la multiplicación entre $H \cdot y'$ en este caso la respuesta es el código 011, que en decimal es el número 3, esto indica que el tercer *bit* de y' es erróneo. Al conocer el *bit* erróneo se procede a cambiar el valor del *bit* y recuperar la palabra en código válido 1101001, por último se toman los primeros cuatro *bits* recuperando de esta manera la palabra de código enviada por el emisor 1101. El producto $H \cdot y$ se llama el síndrome (o síndrome de error) del vector "y". Si el síndrome es cero, "y" es una palabra de código válido; caso contrario el síndrome indica por medio de un número del 1 hasta el 7, la posición en la cual se encuentra el *bit* con error en y para que el mismo sea corregido y recuperar la palabra de código válido por medio de Hamming (Torres, 2014).

3.8 Pruebas de codificación y decodificación en lenguaje Python mediante Modbus.

Para realizar las pruebas de comunicación entre el codificador y el decodificador en Python se necesitan algunas librerías adicionales, las cuales se describen a continuación:

Librería PySerial.

Esta librería se utiliza para trabajar con el puerto serial, permite la configuración de los parámetros del puerto como son: *bits* de paridad, el puerto COM, velocidad de comunicación, etc.

Librería Numpy

Numpy adiciona mayor soporte para vectores y matrices, por lo cual brinda funciones matemáticas de alto nivel para trabajar con vectores o matrices.

Librería *random*

Esta librería se utiliza para generar un número aleatorio en un rango determinado.

Librería sys

La librería "sys" proporciona información acerca de constantes, funciones y procedimientos del intérprete de Python.

Librería *time*

Esta librería proporciona diversas funciones relacionadas con el tiempo.

La librería que se utiliza para la comunicación mediante Modbus es la Modbus-tk que está descrita anteriormente.

3.9 Desarrollo del programa de codificación del código Hamming en lenguaje Python.

El programa codificador hace el papel de maestro, ya que este genera los datos y los envía a los registros que se encuentran en el decodificador (esclavo).
Para probar la codificación del código Hamming (7, 4), se generan aleatoriamente 10 datos de 4 *bits* cada uno, luego se codifican y por último se introduce el error en un solo *bit* de manera aleatoria. Con esto se obtiene un valor de 7 *bits* de los cuales: 4 *bits* representan el dato y 3 *bits* representan el código de redundancia. Al final el programa guarda los datos codificados con error en los registros del esclavo. Los datos se guardan en representación decimal. En la figura 3.5 se muestra el flujograma para el codificador.

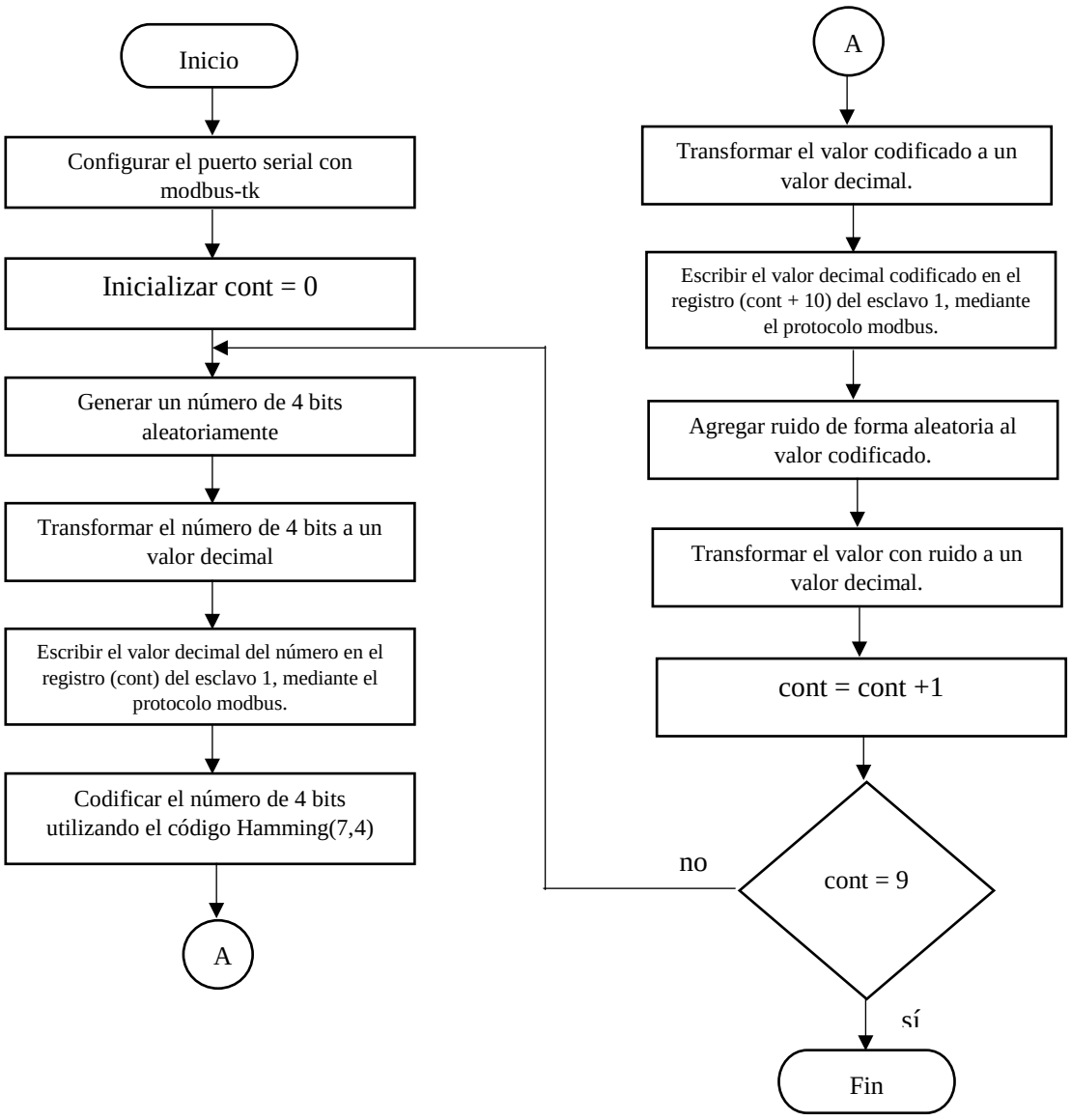


Figura 3.5: Flujograma del codificador Hamming (7,4) con protocolo Modbus.
Fuente: (Torres, 2014).

3.9.1 Implementación del Algoritmo.

La implementación del algoritmo del codificador Hamming (7, 4) en Python y comunicación con protocolo Modbus se encuentra en el Anexo 1.

3.10 Desarrollo del programa de decodificación del código Hamming en lenguaje Python.

El programa decodificador funciona como esclavo. Cuando se inicia el programa entra en modo de espera hasta que el maestro envíe todos los datos codificados. A continuación se realiza la lectura de los registros para proceder a la decodificación de

los datos recibidos. La decodificación se realiza de la siguiente manera: primero se transforma el valor recibido de representación decimal a binario de 8 bits, a continuación se transforma el dato de 8 a 7 bits. Luego se realiza la decodificación del dato para corregir, en caso de que exista el error el programa calcula la posición del mismo y lo corrige. Finalmente se eliminan los bits de redundancia para obtener el dato de manera correcta (sin error).

En la figura 3.6 se muestra el diagrama de flujo del decodificador.

3.10.1 Implementación del Algoritmo.

La implementación del algoritmo del decodificador Hamming (7, 4) en Python y comunicación con protocolo Modbus se encuentra en el Anexo 2.

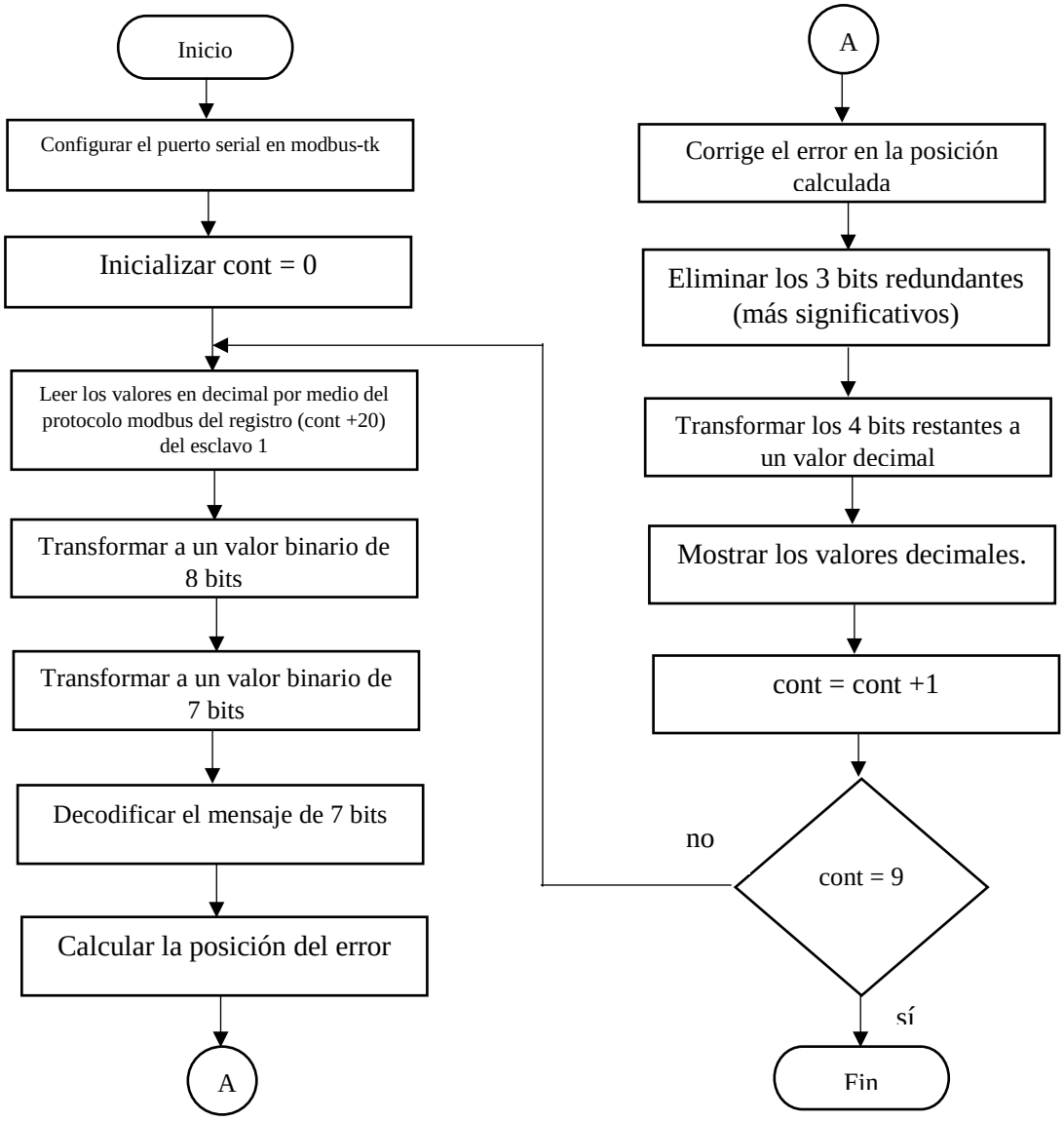


Figura 3.6: Flujograma del decodificador Hamming (7,4) con protocolo Modbus.
Fuente: (Torres, 2014).

3.11 Pruebas de funcionamiento de la comunicación entre el codificador y decodificador Hamming en lenguaje Python.

Para realizar las pruebas entre el codificador y decodificador, el programa codificador (maestro) se encuentra en la PC mientras que el programa decodificador (esclavo) se encuentra en el PcDuino. La comunicación es inalámbrica y se realiza mediante los módulos Xbee.

En la figura 3.7 se muestran los equipos conectados y listos para probar la comunicación entre el decodificador y el codificador.

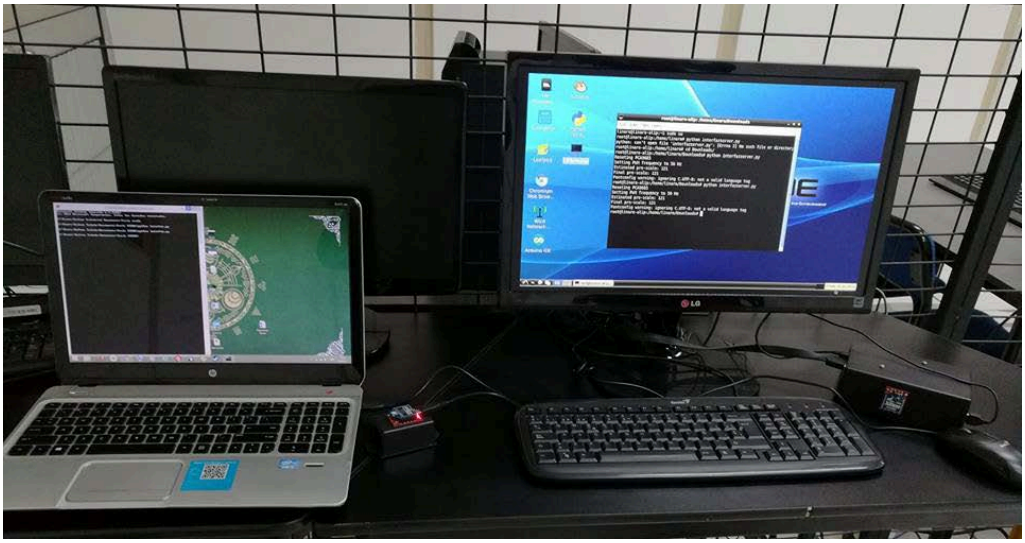


Figura 3.7: Pruebas de funcionamiento entre el codificador y decodificador mediante comunicación inalámbrica.

Para el correcto funcionamiento de los programas, primero se debe iniciar el decodificador y seguidamente se puede iniciar el codificador.

En la figura 3.8 se muestra la pantalla cuando se inicia el decodificador. El esclavo queda en modo de espera hasta que el maestro le envíe los datos.

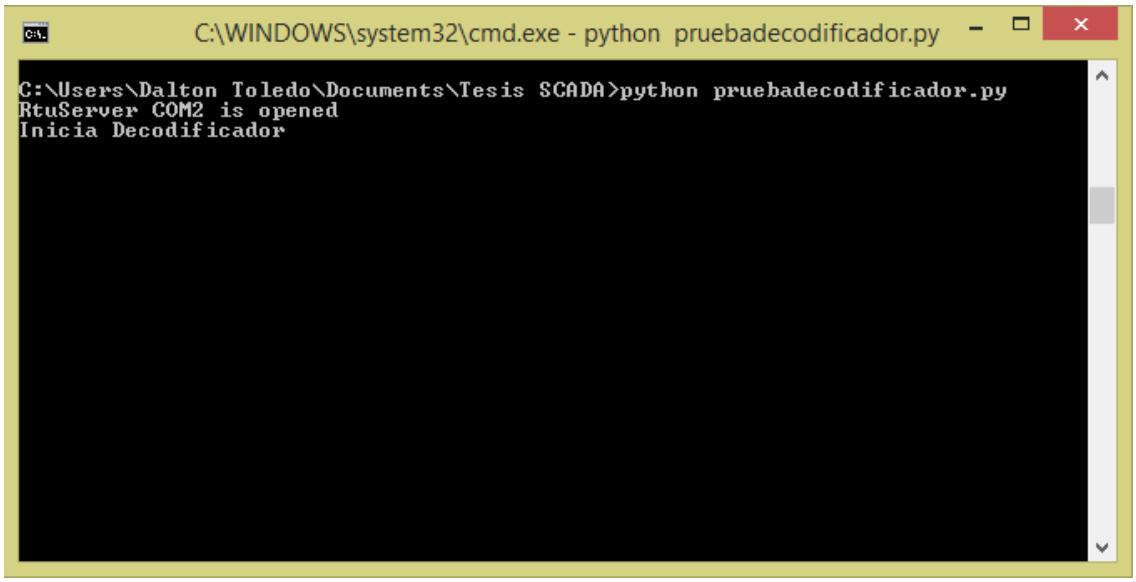


Figura 3.8: Pantalla de inicio del decodificador.

A continuación se inicia el codificador, en la figura 3.9 se muestran todos los datos que son enviados al decodificador.

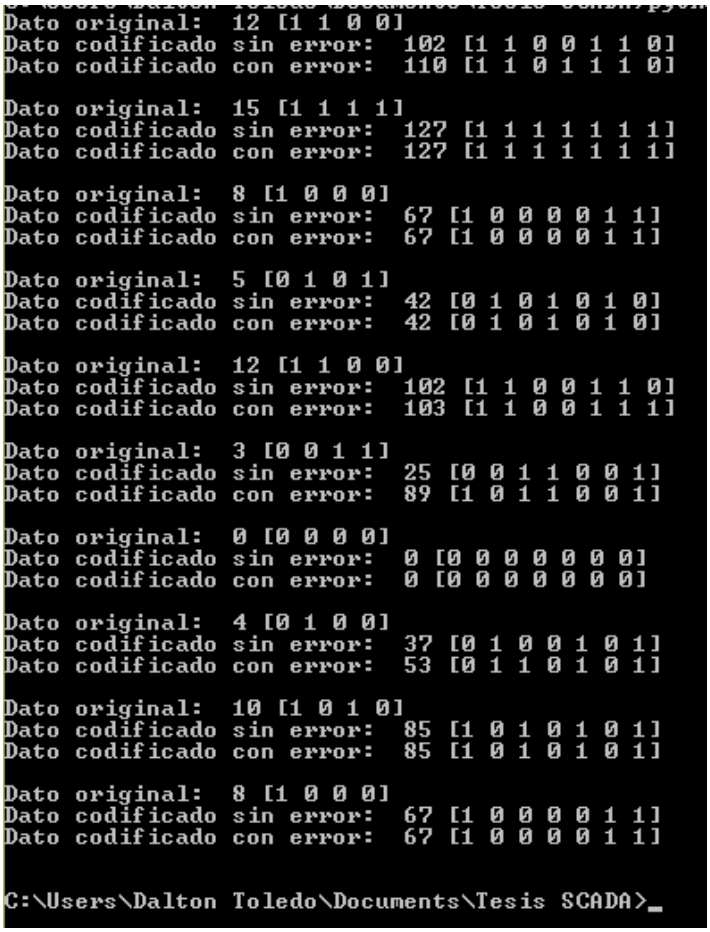


Figura 3.9: Pantalla del codificador, datos que son enviados el decodificador.

Cuando se termina la recepción de datos en el decodificador, el programa muestra en la pantalla los valores codificados con error, los valores decodificados sin error y por último el dato original en decimal y binario.

En la figura 3.10 se muestra la pantalla del decodificador con todos los datos indicados anteriormente.

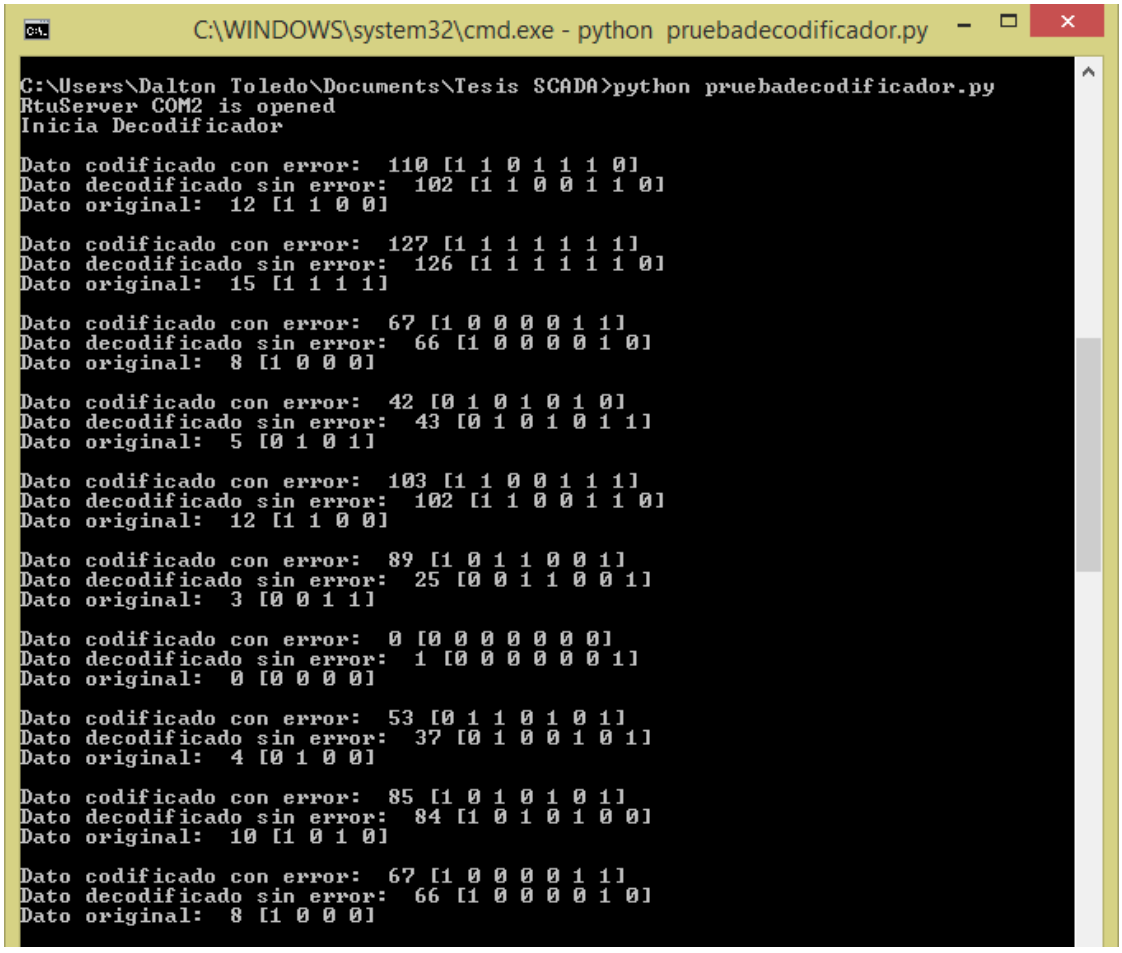


Figura 3.10: Pantalla del decodificador, datos recibidos y decodificados.

Como se puede observar la comunicación entre el codificador y decodificador es correcta ya que se muestra una correspondencia entre los valores codificados y decodificados.

3.12 Conclusiones.

La librería Modbus-tk para Python brinda muchas facilidades al momento de implementar el protocolo Modbus para la comunicación entre el maestro y el esclavo. Otra librería importante para la comunicación es la librería PySerial, está nos permite

tener acceso al puerto serial configurando los diferentes parámetros del mismo (velocidad de transmisión, *bits* de paridad, número de puerto, *bits* de parada).

Existen varios métodos para la detección y corrección de errores, entre ellos está el código Hamming. El código Hamming es una técnica de corrección de errores donde la corrección se realiza en el dispositivo receptor mediante códigos de redundancia que envía el dispositivo emisor. La gran ventaja de esta técnica es que no se pierde tiempo en la retransmisión de datos, si estos tienen error. Debido a estas ventajas el código Hamming se implementa para la corrección y detección de errores en este proyecto.

CAPÍTULO 4

IMPLEMENTACION DEL SISTEMA SCADA PARA LA COMUNICACIÓN MODBUS ENTRE EL CODIFICADOR Y DECODIFICADOR HAMMING

4.1 Introducción.

Los sistemas SCADA son utilizados ampliamente en la industria para la supervisión, control y adquisición de datos en los procesos industriales, en los últimos años los sistemas SCADA han progresado de manera sustancial en términos de funcionabilidad, escalabilidad, rendimiento y apertura, tanto así que es posible realizar procesos muy complicados, los cuales se los puede aplicar en la industria que requiera de este tipo de automatización (Penin A. R., Sistemas SCADA, 2013).

SCADA son las siglas de *Supervisory Control And Data Acquisition* o en español Supervisión, Control y Adquisición de Datos, como su nombre lo indica es un sistema de control que supervisa una planta o proceso por medio de una estación maestra, para el *hardware* generalmente se utilizan PLCs u otros dispositivos de control (A. Daneels, 1999).

La distribución física de un sistema SCADA variará dependiendo de las características que requiera cada aplicación.

En este capítulo se describen las características principales de un sistema SCADA para el desarrollo y la implementación de la comunicación Modbus entre el codificador y decodificador Hamming, con el objetivo de mover un manipulador de 3GDL.

4.2 Descripción del sistema SCADA.

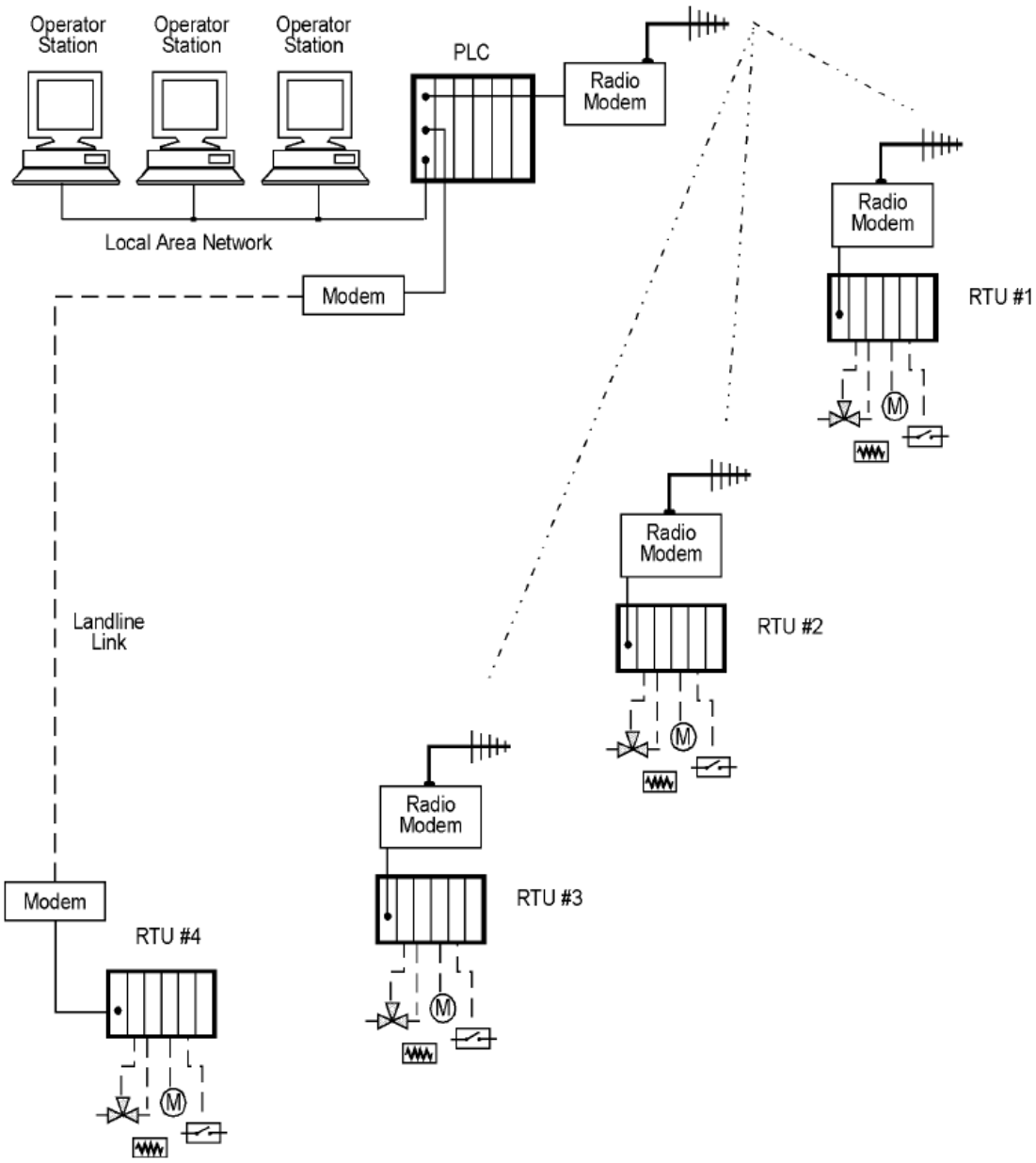


Figura 4.1: Diagrama de un sistema SCADA típico.
Fuente: (Bailey & Wright, 2013).

En un sistema SCADA complejo existen esencialmente cuatro niveles o jerarquías:

1. Sistema de control distribuido.
2. Terminales de ordenación y RTUs.
3. Sistemas de comunicaciones.
4. Estación maestra o estaciones maestras.

4.2.1 *Distributed control system* (DCS) Sistema de Control Distribuido

En un DCS (ver figura 4.2), la adquisición de los datos y el control de las funciones son realizadas por un número de unidades basadas en microprocesador situadas cerca de los dispositivos a controlar o del instrumento del cual se necesita leer los datos. Los sistemas DCS han evolucionado en sistemas que proveen un control analógico muy sofisticado. Se proporciona un conjunto estrechamente integrado de interfaces de operador para permitir configuraciones sencillas del sistema y control del operador. Las velocidades para el flujo de datos son bastante altas usualmente desde 1 Mbps hasta los 10Mbps.

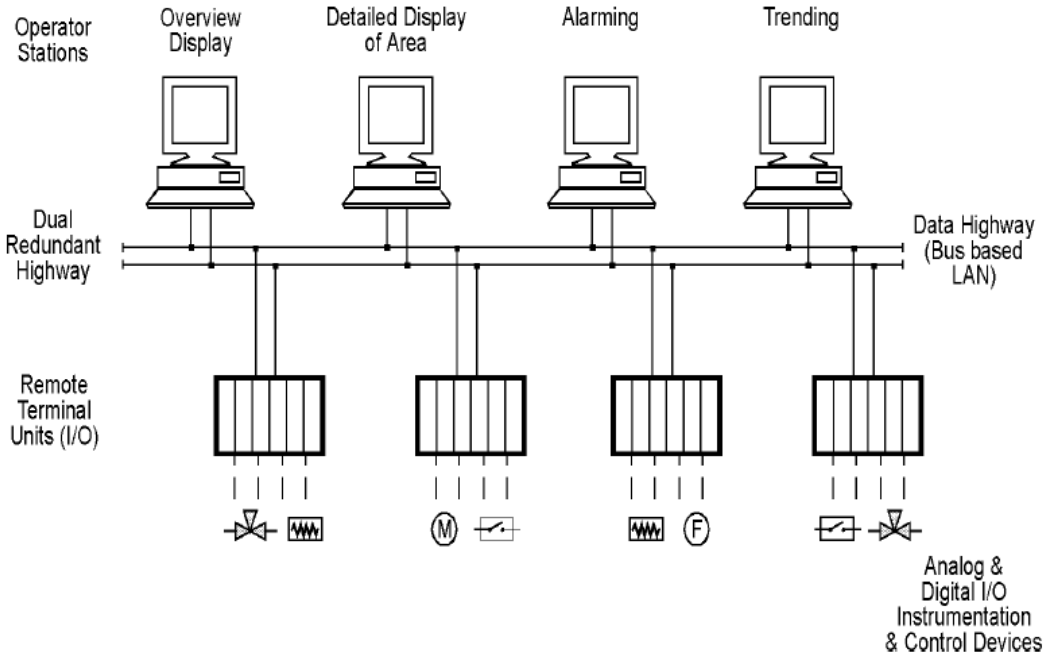


Figura 4.2: *Distributed control system* (DCS).
Fuente: (Bailey & Wright, 2013).

4.2.2 Terminales de ordenación y RTUs

4.2.2.1 *Programmable logic controller* (PLC) Controlador Lógico Programable.

Desde los años 1970, los PLCs han reemplazado a los relés cableados con una combinación de *software* de lógica de escalera y módulos electrónicos de entrada y salida de estado sólido, son usados a menudo en la implementación de un SCADA

RTU ya que ofrecen una solución de *hardware* estándar, la cual tiene un precio económico (ver figura 4.3).

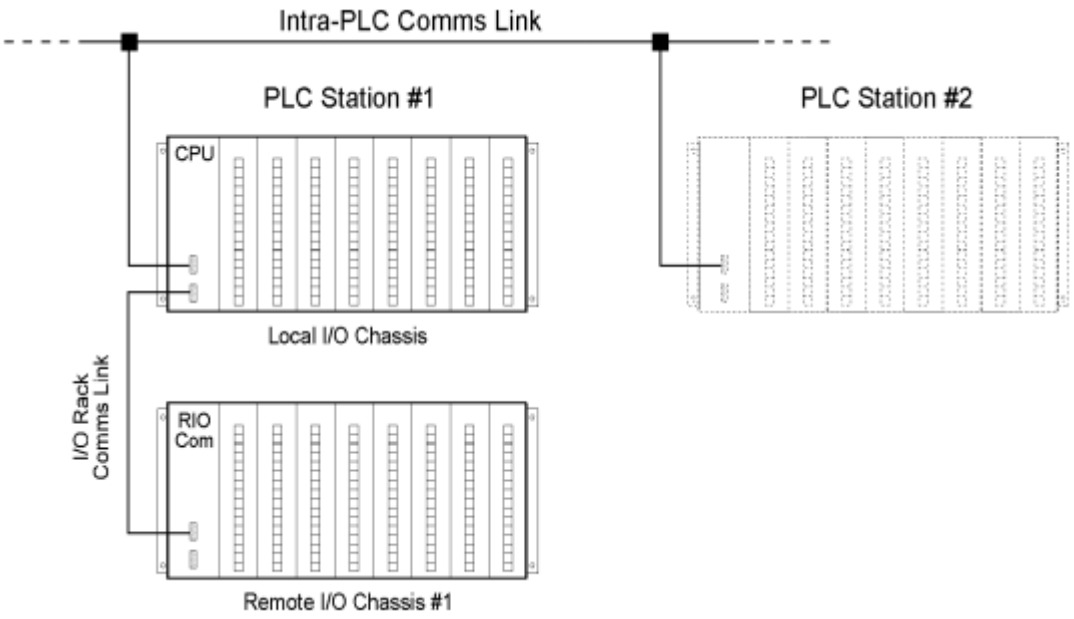


Figura 4.3: Programmable Logic Controller (PLC) system.
Fuente: (Bailey & Wright, 2013).

Se considerará a los DCS, PLC e instrumentos inteligentes como variaciones o componentes del concepto básico de SCADA.

4.2.2.2 *Remote Terminal Units* (RTU) Unidad Terminal Remota

Una RTU como el título implica, es una unidad de adquisición y control de datos autónomos, que monitorea y controla el equipo en una ubicación remota desde la estación central. Su principal tarea es controlar y adquirir datos de equipos de proceso en la ubicación remota y transferir estos datos a una estación central. Generalmente tiene la facilidad de tener sus programas de configuración y control descargados dinámicamente desde alguna estación central. Existe una instalación que puede ser configurada localmente por alguna unidad de programación RTU, aunque tradicionalmente la RTU se comunica de vuelta a una estación central, también es posible comunicarse sobre una base punto-a-punto, las RTUs de pequeño tamaño generalmente tienen menos de 10 a 20 señales analógicas y digitales, las RTUs de tamaño medio tienen 100 entradas digitales y 30 a 40 entradas analógicas. (Bailey & Wright, 2013).

Una configuración típica de una RTU se muestra en la figura 4.4:

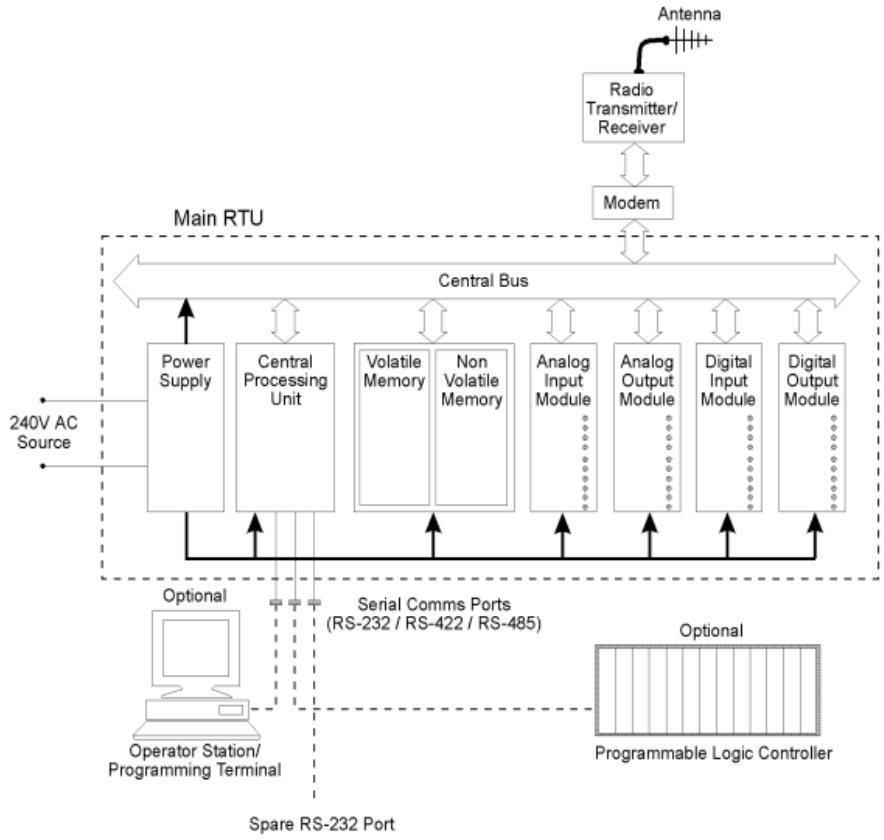


Figura 4.4: Estructura de *hardware* típico de una RTU.
Fuente: (Bailey & Wright, 2013).

4.2.3 Sistemas de comunicación del SCADA

El sistema de comunicaciones proporciona el camino para las comunicaciones entre la estación maestra y los sitios remotos. Este sistema de comunicación puede ser radio, línea telefónica, microondas y hasta posiblemente satelital. Protocolos de comunicación específicos y diferentes filosofías para la detección de errores son usados para transmitir los datos de manera óptima y eficiente.

Los sistemas SCADA dependen en gran medida de la efectividad de transmisión de datos entre las RTUs y las estaciones maestras, para realizar este tipo de comunicación se requiere de un protocolo de transmisión de datos,

4.2.4 *Master Terminal Unit* (MTU) Estación maestra

La estación maestra recolecta datos de los RTUs existentes y generalmente proporcionan una interfaz de operación para presentar información y control de los sitios remotos. La tecnología SCADA ha existido desde los principio de los años sesentas y ahora existen dos enfoques posibles, *Distributed Control System* (DCS) y

Programmable Logic Controller (PLC). En adición, existe una tendencia creciente a usar instrumentos inteligentes como el componente clave en todos estos sistemas.

4.2.5 Consideraciones y beneficios de un sistema SCADA

Las consideraciones típicas cuando se desarrolla un sistema SCADA son:

- Requisitos generales de control.
- Lógica secuencial.
- Control de lazo analógico.
- Relación y número de puntos analógicos a digitales.
- Velocidad de control y adquisición de datos.
- Estaciones de control maestro / operador.
- Tipos de visualización.
- Requisitos de archivos históricos.
- Consideración del sistema.
- Confiabilidad / disponibilidad.
- Velocidad de las comunicaciones / tiempo de actualización / velocidades de exploración del sistema.
- Redundancia del sistema.
- Capacidad de expansión.
- *Software* de aplicación y modelado.

Algunas de las razones típicas para implementar un sistema SCADA son:

- Mejora de operación de la planta o del proceso que resulta en un ahorro debido a la optimización del sistema.
- Mayor productividad del personal.
- Mejora de la seguridad del sistema gracias a una mejor información y un mejor control.
- Protección del equipo de la planta.
- Salvaguardar el ambiente de un fallo del sistema.
- Mejora de los ahorros de energía gracias a la optimización de la planta.
- Recepción mejorada y más rápida de datos para que los clientes puedan ser facturados con mayor rapidez y precisión.

- Regulaciones gubernamentales para la seguridad y medición de gas (por regalías e impuestos, etc.)

4.3 Diseño del sistema SCADA.

Para el diseño del sistema SCADA se utiliza dos herramientas de *software*:

1.- Blender

2.- Qt designer.

4.3.1 Blender

4.3.1.1 Introducción

Blender es un *software* para la creación de contenidos en 2D y 3D (ver figura 4.5), lanzado al público en 1994, fue reprogramado entre el 2008 y el 2010 para mejorar sus funciones, flujo de trabajo e interfaz, este ofrece una gran variedad de funcionalidad para el texturizado, modelado, iluminación, animación y post-procesado de video, debido a su arquitectura abierta ofrece extensibilidad con otros *software* como Python (Williamson, 2012).

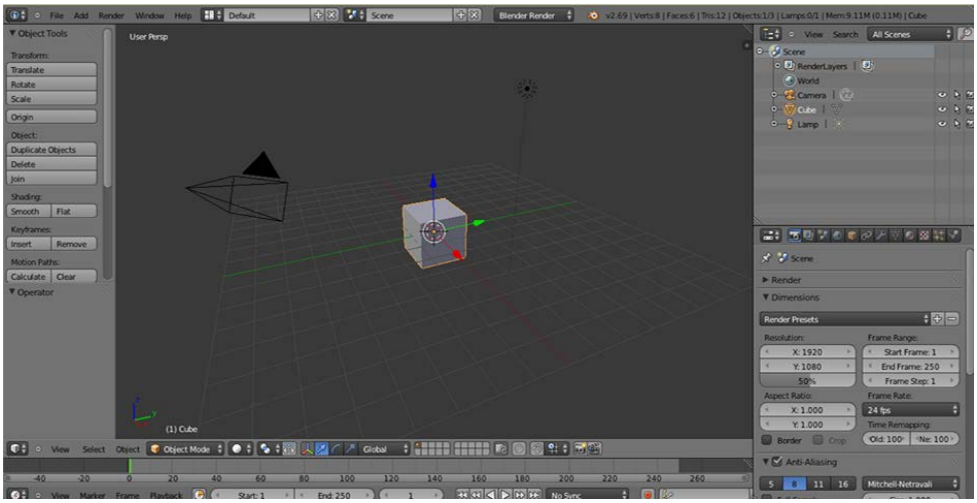


Figura 4.5: Área de trabajo de Blender.

4.3.1.2 Animación y codificación en Blender

Blender permite la interacción de los objetos en 3D y código Python, el brazo robótico se diseñó en función de varios objetos en 3 dimensiones, por ejemplo:

La base del robot es un prisma en donde se ubica la primera articulación en forma de un cilindro, seguido de este tenemos el primer eslabón en forma de un prisma alargado,

las siguientes articulaciones y eslabones se arman uno tras de otro como se observa en la figura 4.6.

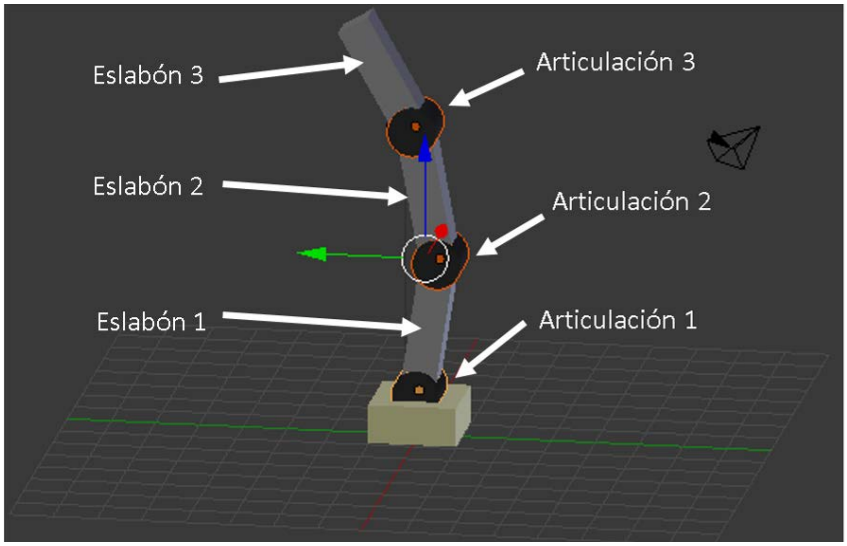


Figura 4.6: Brazo robótico de tres grados de libertad en 3D.

Como se puede observar en la figura 4.7 el objeto diseñado se asemeja a un brazo robótico de 3GDL, donde se puede manipular las 3 articulaciones para variar los ángulos del manipulador.

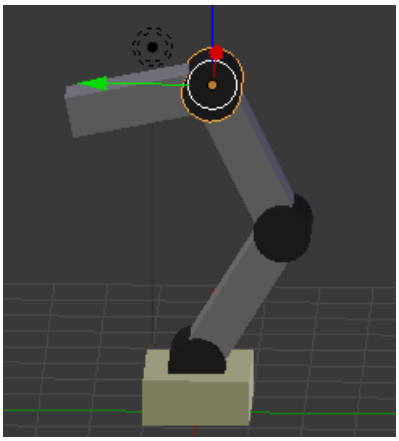


Figura 4.7: Ejemplo de movimiento del brazo simulado.

Para la interacción entre la simulación del brazo y el brazo real, se obtiene la rotación de cada articulación en el brazo simulado de manera que cuando exista movimiento de cualquiera de las articulaciones (figura 4.7), el código detecta automáticamente el número de articulación y de cuanto fue la rotación, después envía los datos para que estos sean entregados al esclavo y realice los movimientos. Para enviar los datos

realizados se emplea el código Python, cada vez que una articulación varia el ángulo envía la nueva posición al codificador y este envía al esclavo.

4.3.2 Qt Designer

4.3.2.1 Introducción

El *software* de código abierto QT fue creado por dos ingenieros noruegos Haavard Nord y Erick Chanble-Eng, por la necesidad de disponer de un GUI[33] para la aplicación de C++, a partir de 2000 la compañía "Trolltech" lanza la licencia gratuita para el desarrollo del *software* libre, QT proporciona varios elementos y herramientas gráficas para la creación de interfaces y aplicaciones multiplataforma que puede ser enlazado a Python (Guiérrez, 2008).

4.3.2.2 Codificación y diseño en la interfaz gráfica para Python.

Para el desarrollo de la interfaz gráfica en Python se utiliza la herramienta de diseño Qt Designer (ver figura 4.8). Esta herramienta facilita la tarea de construir ventanas teniendo la posibilidad de arrastrar botones, cuadros de texto, etiquetas y con la opción de modificar las diferentes propiedades de dichos objetos.

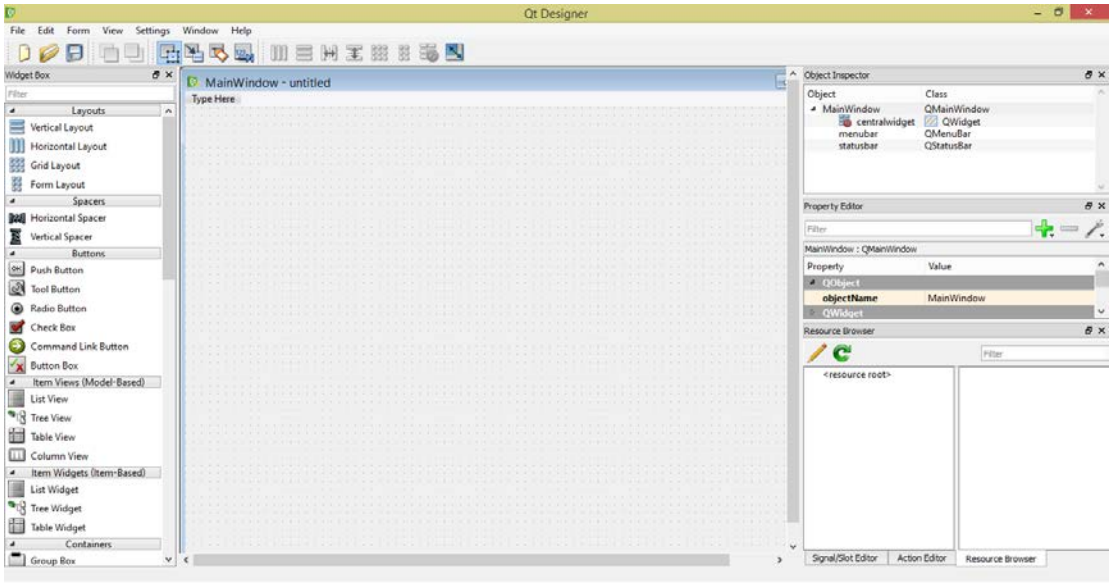


Figura 4.8: GUI del Qt Designer.

Como parte del sistema SCADA el proyecto tiene dos partes: primero el maestro que consta de la interfaz gráfica en Python y el segundo la adquisición de datos del esclavo

[33] GUI: *Graphical User Interface*

por parte del sistema SCADA para el respectivo análisis del comportamiento del movimiento de la trayectoria del manipulador de 3GDL.

La interfaz gráfica del maestro se observa en la figura 4.9, mientras que el esclavo en la figura 4.10.

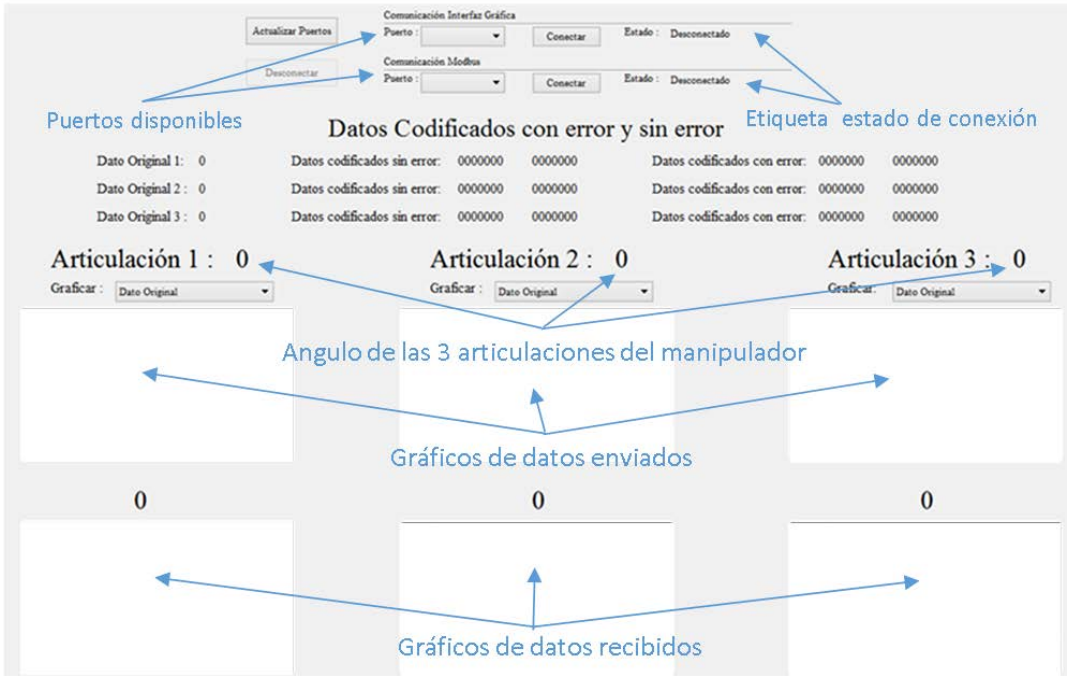


Figura 4.9: Interfaz gráfica del maestro.

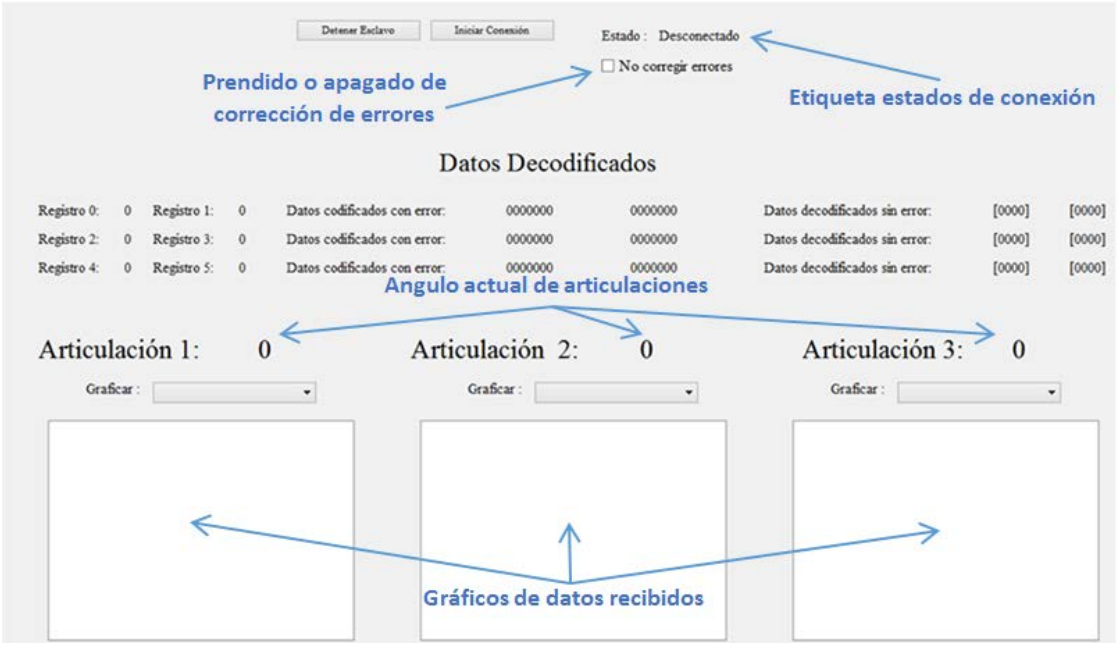


Figura 4.10: Interfaz gráfica del esclavo.

4.4 Implementación del sistema SCADA

4.4.1 Código del Maestro.

El proceso de codificación de los datos recibidos desde la simulación del brazo robótico (SCADA), empieza transformando los datos de las 3 articulaciones representados en decimal a binaria de 8 *bits*, es decir se tienen 3 números binarios.
Para proceder a la codificación del dato de cada articulación el programa divide la trama recibido de 8 *bits* en dos de 4 *bits*, después con los valores obtenidos se realiza la codificación Hamming, que consiste en agregar 3 bits de redundancia para la corrección de errores. Cuando se obtienen los 7 *bits* de los datos codificados, el programa inserta aleatoriamente un error en uno de los 7 *bits* y por último los datos se envían al esclavo.

4.4.1.2 Flujograma

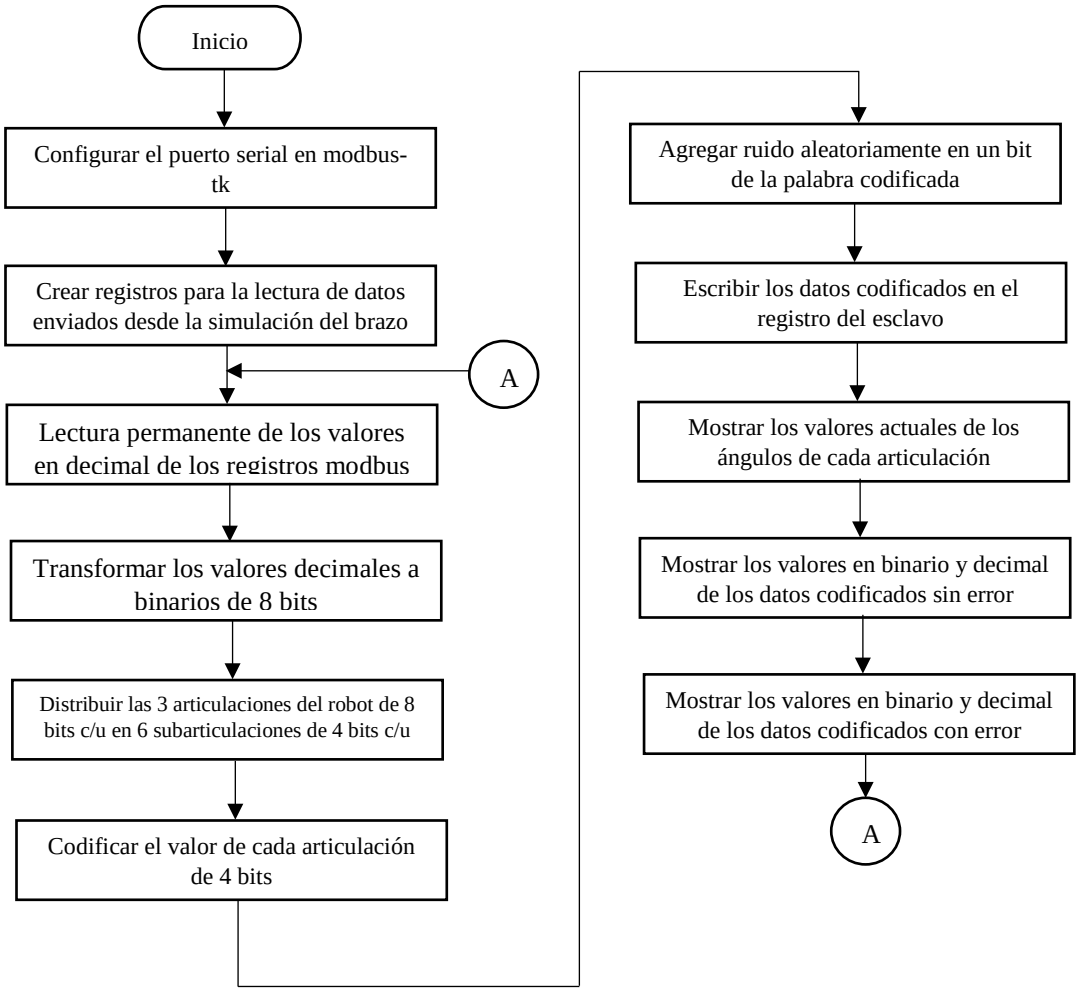


Figura 4.11: Diagrama de flujo del maestro.

4.4.1.3 Implementación del algoritmo principal.

El programa del maestro (Anexo 4) tiene el nombre de interfaz.py, contiene diferentes librerías, clases y funciones descritas en la tabla 4.1:

Tabla 4.1: Funciones codificador.

Librerías	**Clases y Funciones**
Serial	
Os	
Numpy	
Random	
Sys	
Pyqtgraph	
Modbus_tk	
PyQt4	
	desconectarServer
	actualizarPuertos
	timerLectura
	timerGraficos
	cargarPuertos
	conectarPuerto1
	conectarPuerto2
	codificación

4.4.1.3.1 Descripción de las Funciones

DesconectarServer: Es una función del programa principal cuyo objetivo es terminar la conexión con el esclavo.

ActualizarPuertos: Es una función del programa principal la cual enlista todos los puertos disponibles en ese momento.

TimerLectura: Es una función del programa principal que se ejecuta cada 10 ms, una vez que se haya activado el *timer* (en el programa el objeto se llama *ctimer*). Esta función a su vez llama a la función codificación.

TimerGraficos: Es una función del programa principal, se ejecuta cada 10 ms después de la activación del objeto *timer* (en el programa el objeto se llama *gtimer*). Esta función contiene todos los procesos necesarios para mostrar las gráficas.

CargarPuertos: Es una función del programa principal, esta función se encarga de mostrar los puertos enlistados anteriormente.

ConectarPuerto1: Es una función del programa principal, se encarga de realizar la conexión serial entre el codificador (maestro) y el programa de simulación del brazo robótico. Esta función también activa los *timers*, pero únicamente si están establecidas las dos conexiones: 1) entre el maestro y el esclavo, 2) el maestro y el programa de simulación del brazo robótico.

ConectarPuerto2: Es una función del programa principal, se encarga de realizar la conexión serial entre el maestro y el esclavo. Esta función también activa los *timers*, pero únicamente si están establecidas las dos conexiones: 1) entre el maestro y el esclavo, 2) el maestro y el programa de simulación del brazo robótico.

Codificación: Es una función del programa principal, contiene todo el proceso de codificación de los datos en código Hamming, también realiza la inserción del error en la palabra codificada. Después de realizar la codificación los datos, son guardados en los registros de esclavo.

4.4.2 Código del esclavo.

El proceso de decodificación requiere una constante lectura de los registros ModBus para obtener los datos actualizados. Una vez que se tiene los valores decodificados y sin los bits de paridad, el programa procede a juntar los cuatro *bits* decodificados del primer registro con los del segundo, los datos decodificados del tercer registro se unen con los del cuarto registro y por último los datos decodificados del quinto registro se unen con los del sexto registro, obteniendo 3 datos de 8 *bits*, estos al ser transformados a representación decimal se obtienen los datos de las articulaciones sin error para ser enviadas a cada uno de los servomotores del manipulador de 3GDL. Se puede configurar la velocidad de transmisión de los datos enviados al robot según se considere la necesidad del caso.

4.4.2.1 Flujograma

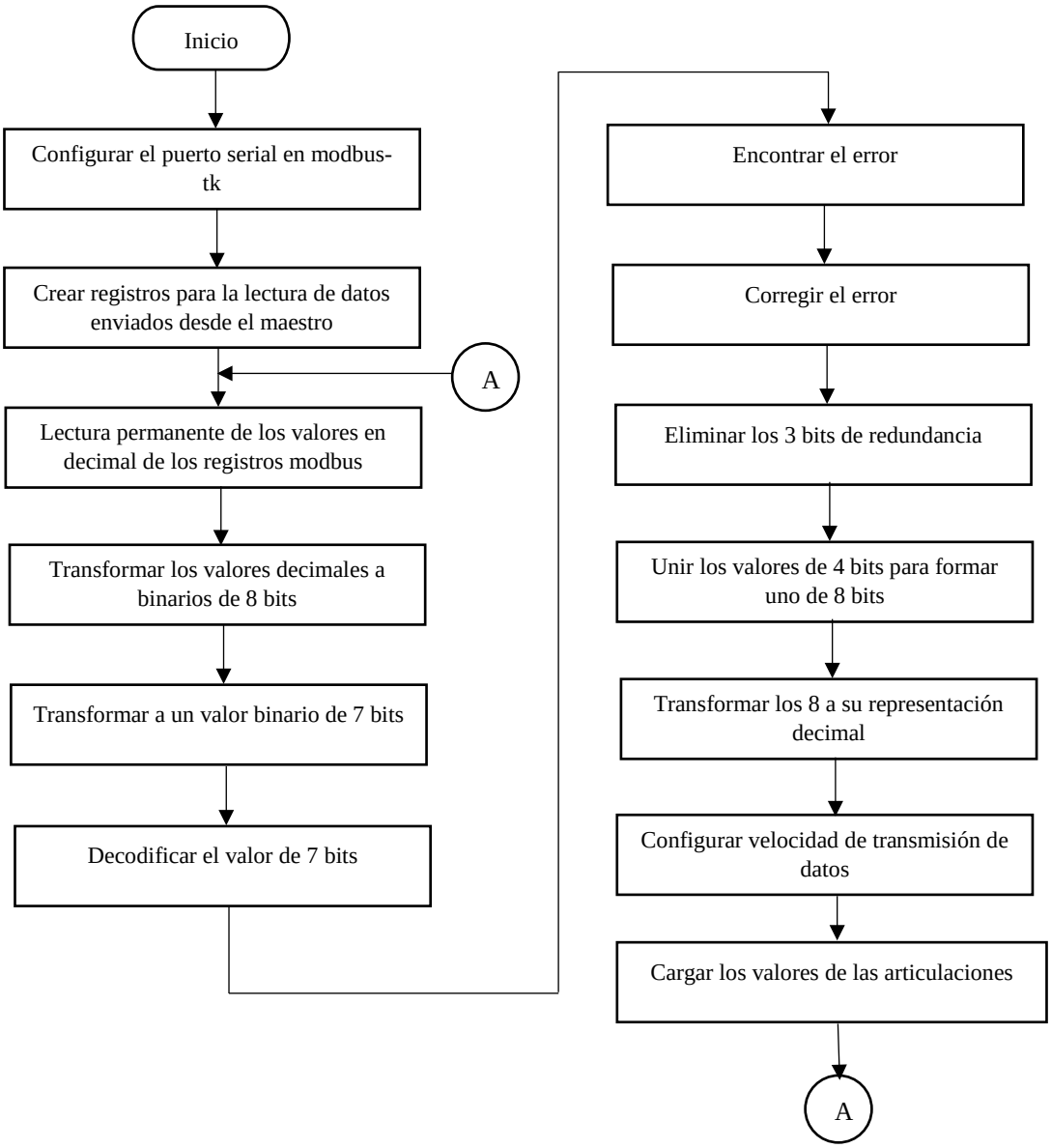


Figura 4.12: Diagrama de flujo del esclavo.

4.4.2.2 Implementación del algoritmo.

El programa del esclavo (Anexo 5) llamado interfazserver.py, contiene diferentes librerías, clases y funciones descritas en la tabla 4.2:

Tabla 4.2: Funciones decodificador.

Librería	**Clases y Funciones**
Serial	
Os	
Numpy	
Random	
Sys	
Pyqtgraph	
Modbus_tk	
PyQt4	
Adafruit_PCA9685	
	timerLectura
	timerGraficos
	desconectarServer
	conectarPuerto
	decodificacion

4.2.2.1 Descripción de las Funciones

TimerLectura: Es una función del programa principal que se ejecuta cada 10 ms, una vez que se haya activado el *timer* (en el programa el objeto se llama *ctimer*). Esta función a su vez llama a la función decodificación.

TimerGraficos: Es una función del programa principal, se ejecuta cada 10 ms después de la activación del objeto *timer* (en el programa el objeto se llama *gtimer*). Esta función contiene todos los procesos necesarios para mostrar las gráficas.

DesconectarServer: Es una función del programa principal cuyo objetivo es terminar la conexión con el maestro.

ConectarPuerto: Es una función del programa principal, se encarga de realizar la conexión serial entre el maestro y el esclavo. Esta función activa los *timers*, y crea los registros Modbus para la lectura de los datos enviados desde el maestro.

Decodificación: Es una función del programa principal, contiene todos los procesos necesarios para la decodificación y corrección de los datos, se encarga también de generar un PMW para cada articulación dependiendo de los valores decodificados.

4.5 Pruebas de comunicación del sistema SCADA

4.5.1 Pruebas en el codificador

Las pruebas del codificador se realizan en base a los siguientes datos:

En el codificador obtenemos los datos de las articulaciones, para las pruebas se muestran los datos codificados sin error y con error (ver figura 4.13).

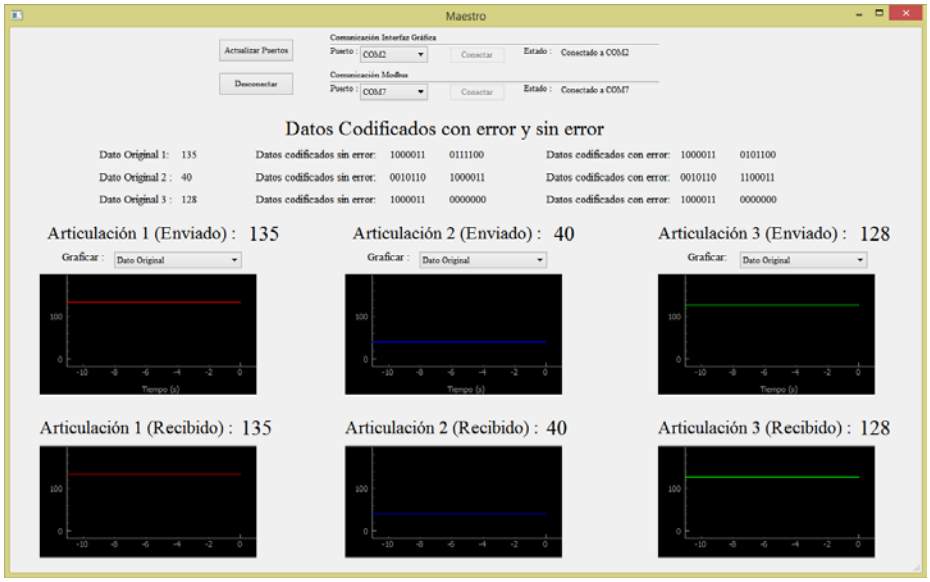
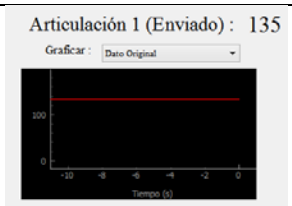
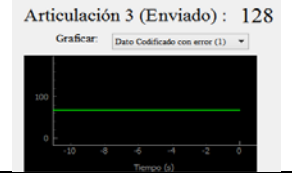


Figura 4.13: Datos originales, codificados sin error y con error.

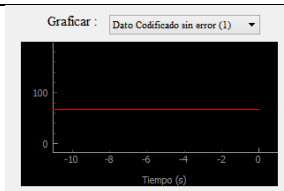
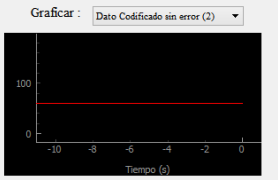
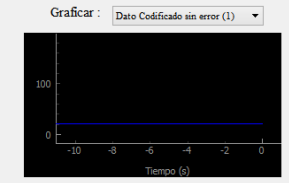
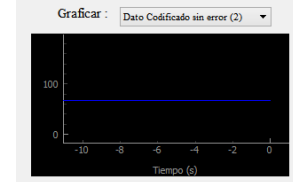
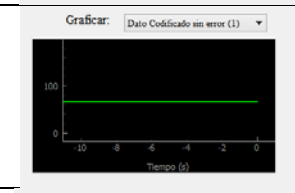
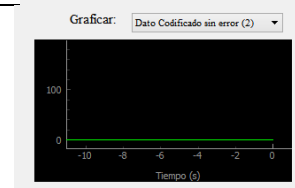
En la tabla 4.3 se muestran los datos originales de las tres articulaciones junto con sus respectivos gráficos.

Tabla 4.3: Datos originales.

Articulación	Datos originales		
	Binario	Decimal	Gráfico
1	10000111	135	
2	101000	40	
3	10000000	128	

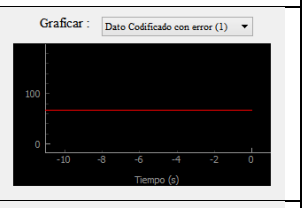
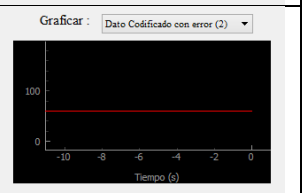
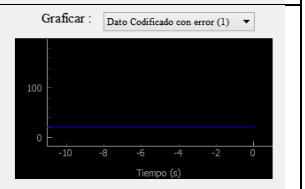
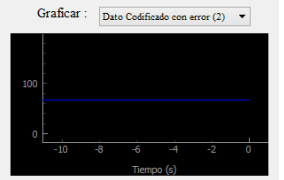
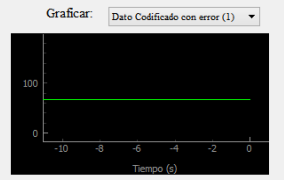
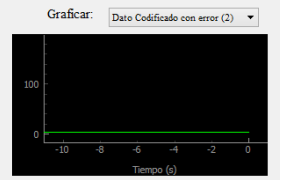
En la tabla 4.4 se muestran los datos codificados sin error en representación binaria y decimal con sus gráficos.

Tabla 4.4: Datos codificados sin error.

Articulación	Datos codificados sin error		
	Binario	Decimal	Gráfico
1	1000011	67	
	0111100	60	
2	0010110	22	
	1000011	67	
3	1000011	67	
	0000000	0	

En la tabla 4.5 se muestran los datos codificados con error en representación binaria y decimal con sus respectivos gráficos.

Tabla 4.5: Datos codificados con error.

Articulación	Datos codificados con error		
	Binario	Decimal	Gráfico
1	1000011	67	
	0101100	44	
2	0010110	22	
	1000011	67	
3	1000011	67	
	0000100	4	

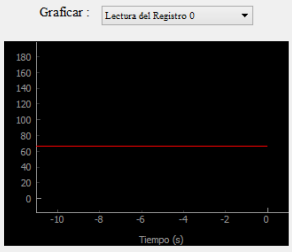
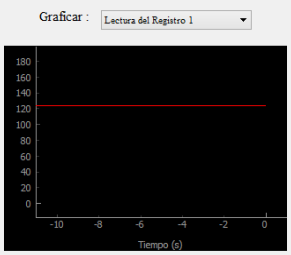
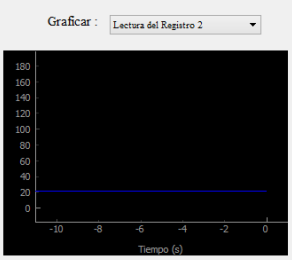
4.5.2 Pruebas en el decodificador

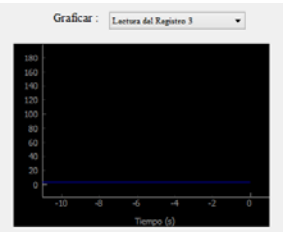
En el decodificador se compara que los ángulos recibidos sean los mismos que los datos originales enviados por el codificador, además visualizamos las gráficas y los datos de los registros. En la figura 4.14 se muestran todos los elementos que contiene el programa.

Figura 4.14: Registros, datos codificados con error y datos decodificados sin error.

En la tabla 4.6 se muestran los valores en los registros y sus gráficos.

Tabla 4.6: Valores codificados con error.

Articulación	**Valores leídos de los registros**		
	Binario	**Decimal**	**Gráfico**
1	1000011	67	
	1111100	124	
2	0010110	22	

	0000011	3	
3	1000011	67	
	0000000	0	

4.6 Conclusiones

4.6.1 Pruebas de comunicación con corrección de errores.

En la parte superior de la figura 4.15 se muestran los gráficos de los datos enviados por el maestro, mientras que en la parte inferior observamos los datos con corrección de errores enviados desde del esclavo, como se puede apreciar los datos enviados son iguales a los receptados.

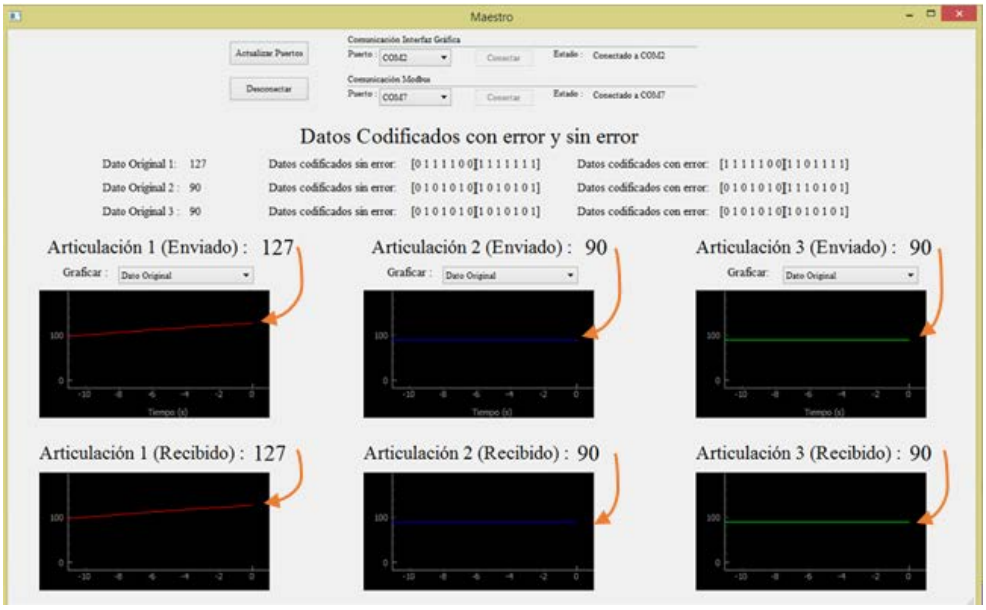


Figura 4.15 : Datos enviados y datos recibidos desde el esclavo.

4.6.2 Pruebas de comunicación sin corrección de errores.

El esclavo tiene una opción para no corregir los errores en los datos decodificados, esto con el fin de mostrar las variaciones que se producen en las tramas cuando no se corrigen los datos recibidos. En la figura 4.16 se muestran los gráficos de los datos enviados desde el maestro y en la parte inferior se visualizan los datos recibidos del esclavo.

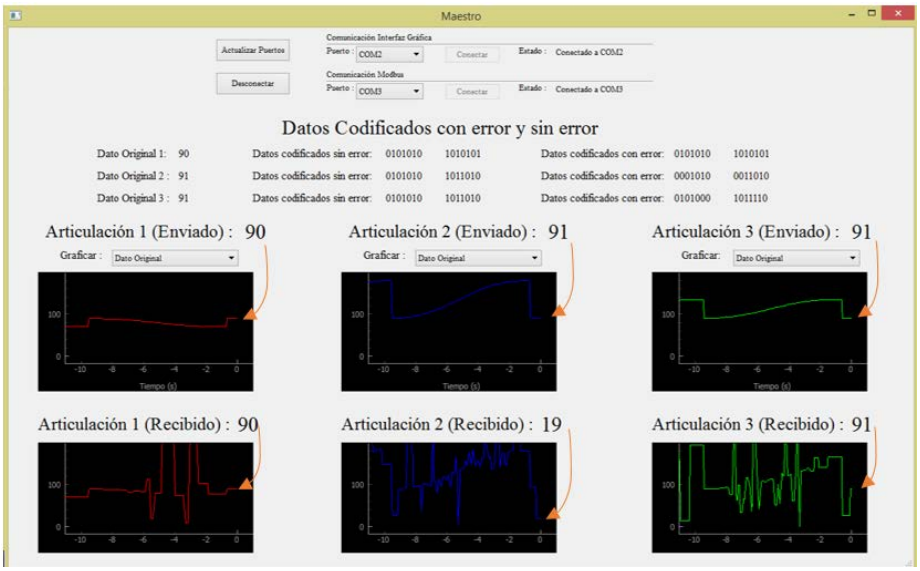


Figura 4.16: Datos enviados y datos recibidos desde el esclavo.

En la figura 4.17 se muestran los gráficos de los datos que recibe el esclavo sin la corrección de errores.

Figura 4.17: Datos decodificados sin corrección de errores.

CAPÍTULO 5

IMPLEMENTACIÓN Y PRUEBAS DE FUNCIONAMIENTO DE UN SISTEMA SCADA MEDIANTE PROTOCOLO MODBUS CON COMUNICACIÓN INALÁMBRICA PARA EL CONTROL DE UN BRAZO DE 3 GDL.

5.1 Introducción

El siguiente capítulo se presenta una breve explicación de los hardwares utilizados para la comunicación del sistema y las diferentes pruebas de funcionamiento realizadas a la comunicación entre dispositivos y finalmente se mostrarán los resultados obtenidos por el trazado de la trayectoria comprobando el correcto funcionamiento de la misma.

5.2 Implementación del sistema

5.2.1 Placa *Expander* Pi.

Mediante la *Expander Pi* se puede extender las capacidades del PcDuino agregando pines de entrada y salida digitales, Conversores Analógico Digital (ADC) y Modulación por Ancho de Pulso (PWM).

La placa *Expander Pi* se caracteriza por tener tres módulos integrados:

1. PCA9685 que se utiliza como PWM y dispone de 8 módulos PWM.
2. MCP3424 que se utiliza como conversor analógico digital y dispone de 4 pines.
3. MCP23016 que se utiliza como entradas y salidas digitales, dispone de 16 pines (Torres, 2014).

En la figura 5.1 se muestran las partes principales de la tarjeta.

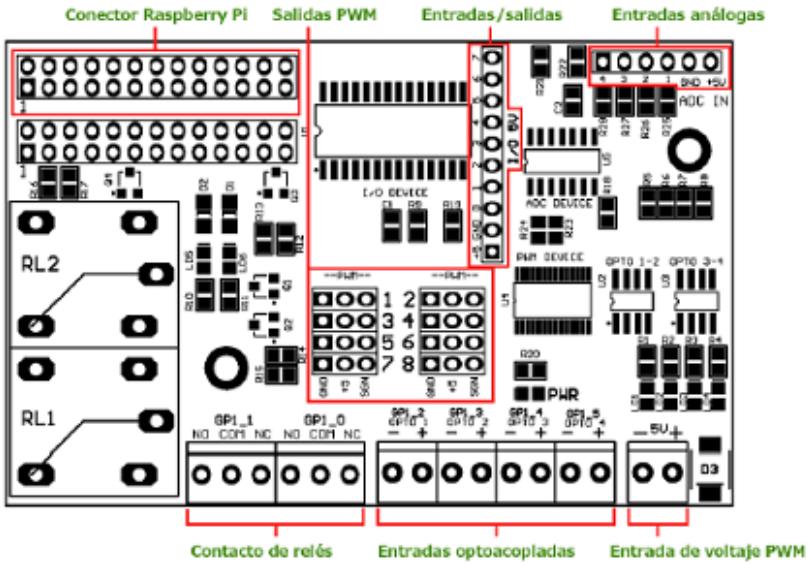


Figura 5.1: Vista de la tarjeta *Expander* Pi.
Fuente: (Torres, 2014).

Contacto de relés: salidas correspondientes a los relés.

Entradas optoacopladas: entrada que soportan hasta 24 VDC.

Entrada de voltaje PWM: sirve para alimentar de forma externa a los PWM.

Salidas PWM: pines de salida PWM con pines de alimentación externa.

Entradas/Salidas: pines configurables como entradas o salidas tolerantes a 5V.

5.2.2 Características del bus I^2C.

La comunicación del bus I^2C es en serie y síncrona. Utiliza dos señales, una marca el tiempo (pulsos de reloj) y la otra es la que se encarga de intercambiar los datos.

Descripción de las señales:

- SCL (*System Clock*) es la línea de reloj que se utiliza para sincronizar las transferencias de datos.
- SDA (*System Data*) es la línea de datos.
- GND (tierra) común para la interconexión entre todos los dispositivos que utilicen en ese momento el bus.
- También puede haber una línea de 5 voltios que alimente a los dispositivos conectados.

En la comunicación I^2C existen dos tipos de dispositivos: maestros y esclavos, solo los dispositivos maestros pueden iniciar una comunicación.

La condición de bus libre o condición inicial, es cuando ambas señales están en estado alto, en este estado cualquier dispositivo maestro puede ocupar el bus estableciendo la condición de inicio. En la condición de inicio el dispositivo maestro pone en estado bajo la línea de datos SDA[34] y la línea de reloj (SCL[35]) permanece en alto.

El primer *byte* transmitido después de la condición de inicio está conformado de la siguiente manera:

- Los primeros siete *bits* indican la dirección del dispositivo con el cual se va a realizar la comunicación
- El octavo *bit* indica la operación que se va a realizar (lectura o escritura).

Para que un dispositivo maestro se comunique con un dispositivo esclavo, se envía a través del bus de comunicación, un *byte* que contiene la dirección del esclavo. Si un dispositivo tiene la dirección enviada en el *byte*, este contesta con el mismo *byte* más un cero al final. A este *bit* se le conoce como *bit* de reconocimiento (ACK) y en estado bajo le indica al dispositivo maestro que el esclavo acepta la solicitud y está listo para comunicarse.

En este punto se establece la comunicación y empieza el intercambio de información entre el maestro y el esclavo (Torres, 2014).

En la figura 5.2 se muestra la conexión entre el PcDuino y el *Expander Pi.*

[34] SDA: *Data Signal*
[35] SCL: *Clock Signal*

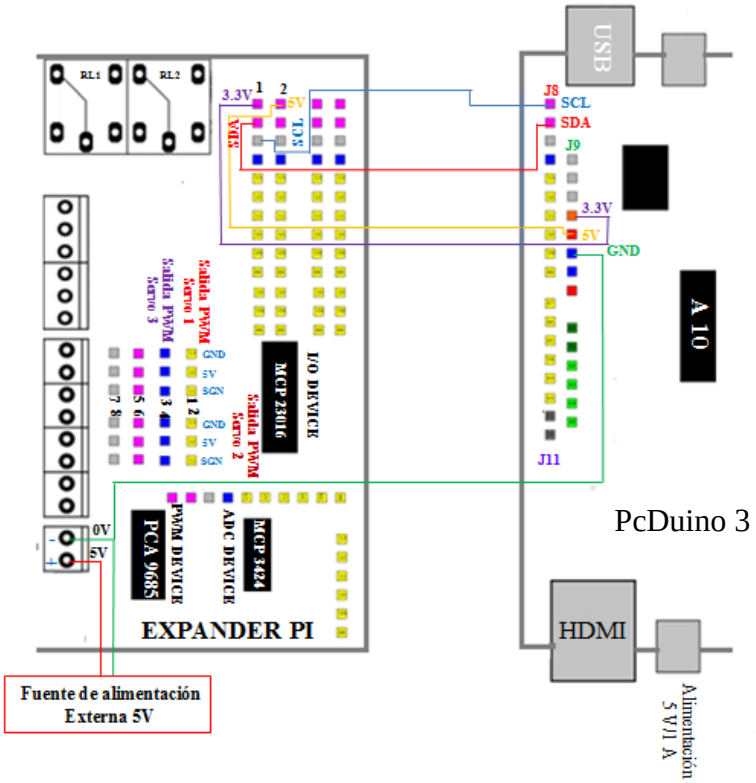


Figura 5.2: Conexión del bus I^2C entre el PcDuino y el *Expander* Pi.
Fuente: (Torres, 2014).

5.2.3 Conexión Expander PI al robot.

Para la conexión entre el manipulador y la placa *Expander* Pi se tiene una fuente de alimentación externa de 5V de 3 amperios, como se observa en la figura 5.3 las articulaciones van conectadas a cada salida de PWM y a su respectiva alimentación.

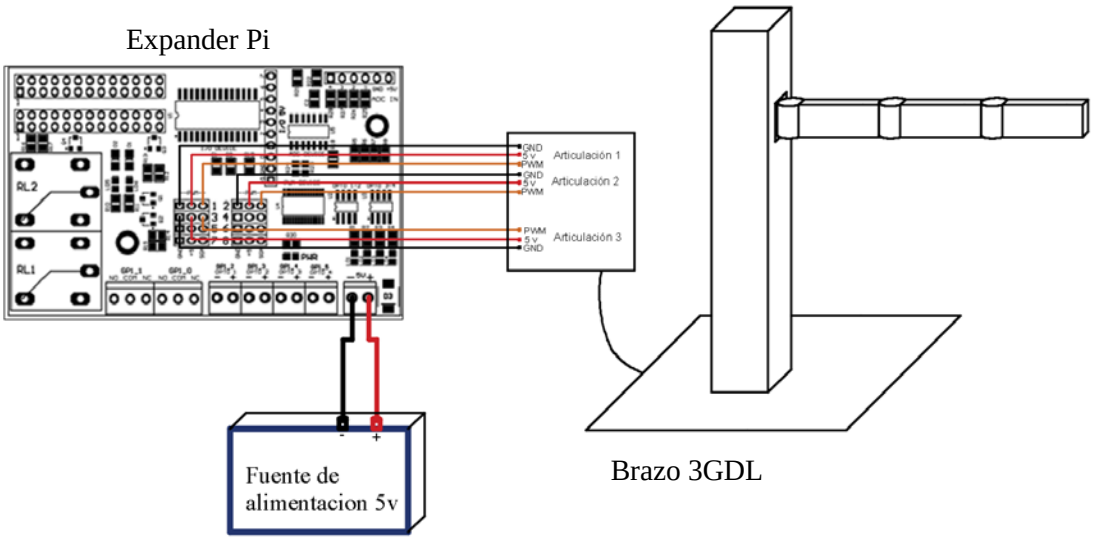


Figura 5.3: Conexión de la placa Expander Pi al robot.

5.2.4 Trayectoria

Para las pruebas del funcionamiento el manipulador de 3GDL sigue la trayectoria mostrada en la figura 5.4. La trayectoria es diseñada por 4 puntos diferentes, estos se diseñan mediante los ángulos de cada eslabón como se observa en la tabla 5.1.

Tabla 5.1 Puntos de la trayectoria en ángulos

	P1	**P2**	**P3**	**P4**
Eslabón 1	180°	135°	45°	0°
Eslabón 2	90°	135°	45°	90°
Eslabón 3	0°	135°	45°	180°

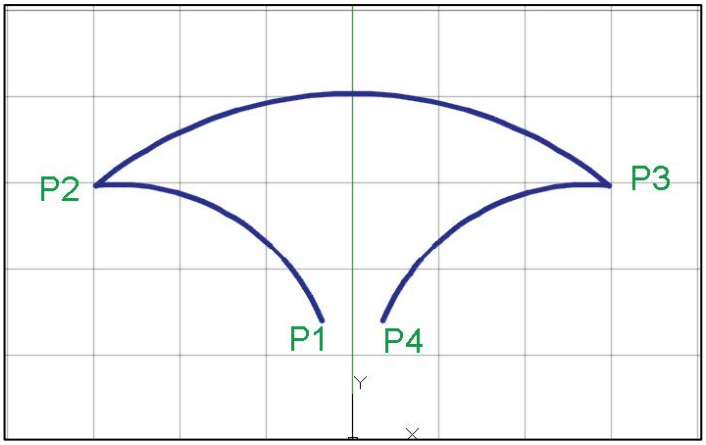


Figura 5.4: Trayectoria del movimiento del manipulador.

5.3 Pruebas de funcionamiento del sistema

Para las pruebas del sistema es necesario contar con los siguientes equipos:

1. Un Pc en el que se encuentra el maestro y el sistema SCADA.
2. Un PcDuino que es utilizado como receptor, es encargado de decodificar el código Haming, y transmitir los datos a la placa *Expander* Pi.
3. Brazo de 3 GDL encargado de realizar los movimientos de las articulaciones de los datos recibidos de la placa *Expander* PI.
4. Los módulos Xbee encargado de la comunicación inalámbrica mediante el protocolo Modbus.
5. Fuente de alimentación para el brazo robótico.

En la figura 5.5 se puede observar todos los elementos necesarios para la prueba de funcionamiento del manipulador de 3Gdl.

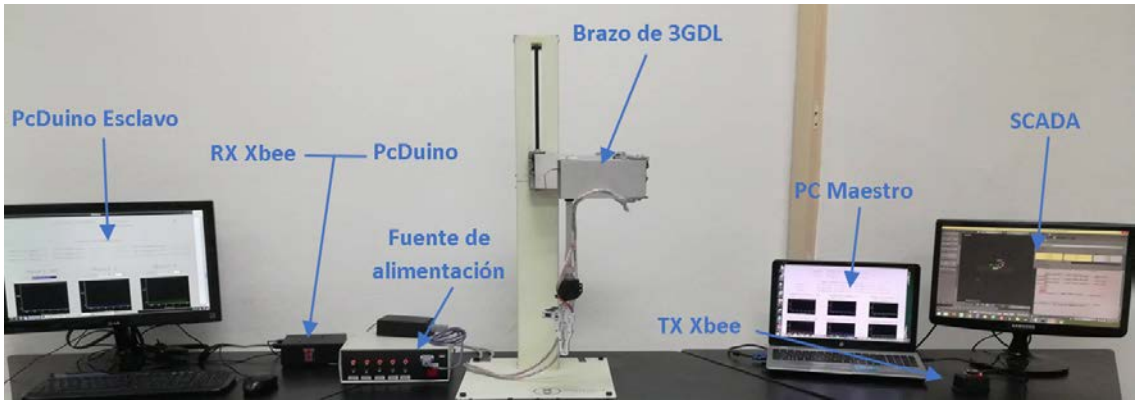


Figura 5.5: Sistema SCADA, maestro esclavo y brazo robótico.

La metodología utilizada para realizar las pruebas de funcionamiento de todo el sistema son las siguientes:

1. Comunicación entre el maestro y esclavo
2. Comunicación entre el esclavo y el brazo robótico
3. Seguimiento de la trayectoria

5.3.1 Pruebas de comunicación entre maestro esclavo.

Para realizar la prueba de comunicación es importante analizar el enlace mediante el protocolo ModBus entre el codificador y decodificador, para este caso enviamos los ángulos 87, 90, 90 como se puede observar en la figura 5.6 desde el maestro al esclavo y comprobamos que los datos enviados sean iguales a los recibidos (ver figura 5.7):

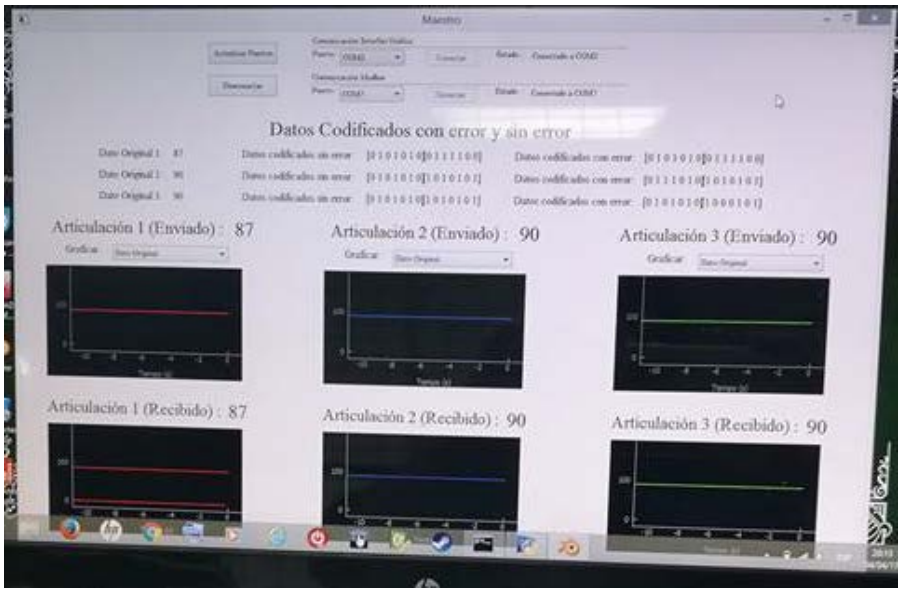


Figura 5.6: datos enviados por el esclavo.

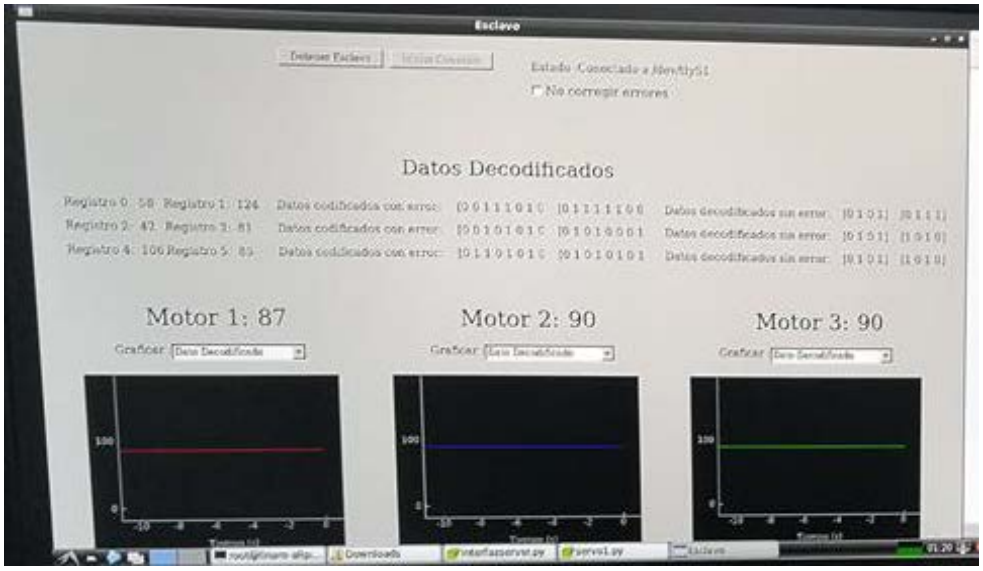


Figura 5.7: Recepción de datos entre esclavo y maestro.

5.3.2 Pruebas de comunicación entre el esclavo y brazo robótico.

Para comprobar que la comunicación sea correcta entre el esclavo y el manipulador iniciamos la conexión del servidor, esto produce que los ángulos recibidos por el esclavo sean transmitidos al brazo robótico, como se observa en la figura 5.8, los ángulos enviados son 87, 90, 90.

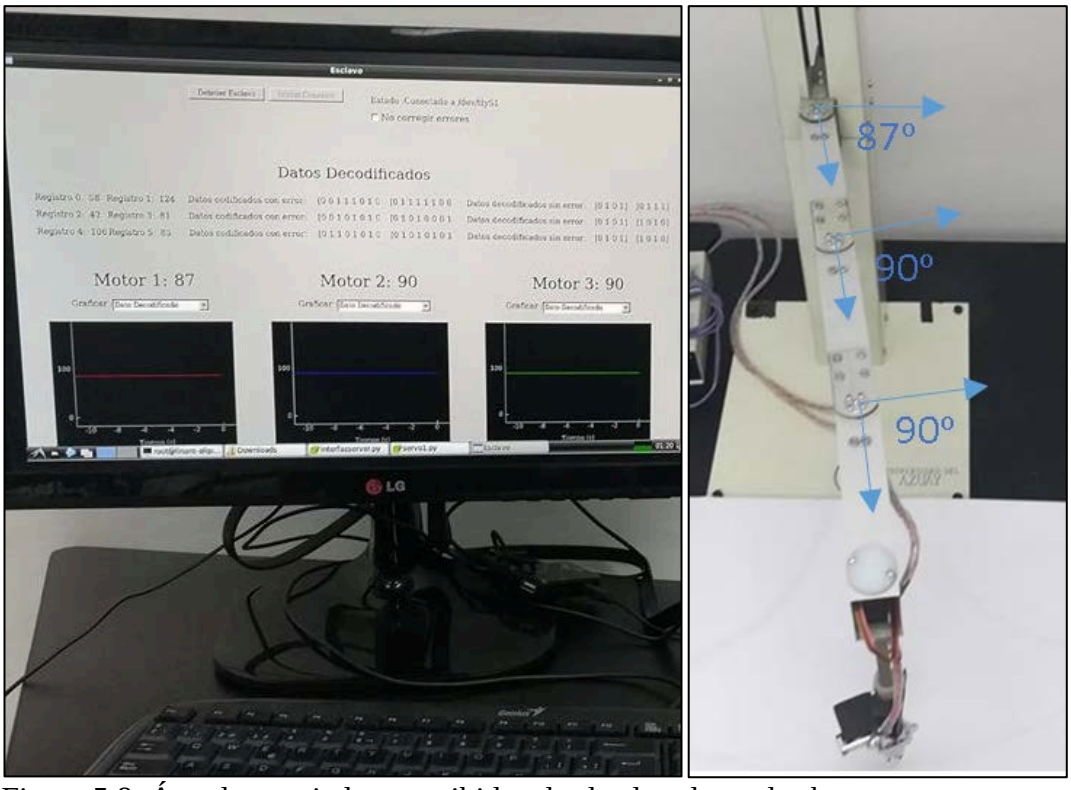


Figura 5.8: Ángulos enviados y recibidos desde el esclavo al robot.

5.3.3 Prueba del seguimiento la trayectoria.

Para las pruebas de la trayectoria se utilizó un láser para poder observar en el papel los datos enviados por el decodificador Hamming.

5.3.3.1 Prueba con error

En esta prueba no se realizó la corrección de errores, como se puede observar en la figura 5.9 el láser está desfasado de la posición de la trayectoria**.**

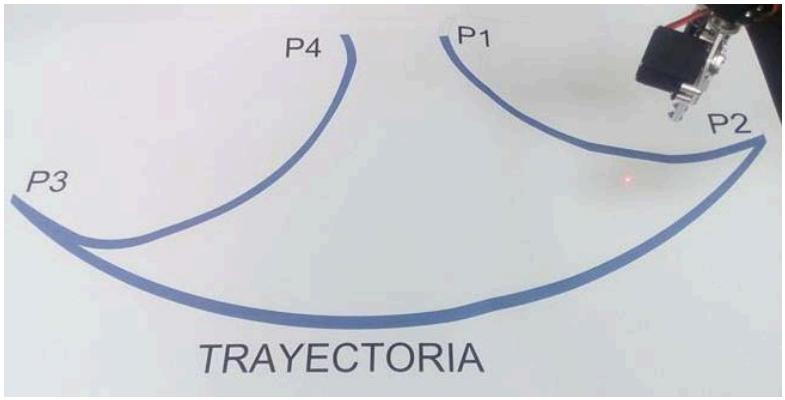


Figura 5. 9: Trayectoria con error.

5.3.3.2 Prueba sin error

En este caso se corrigen los datos enviados por el decodificador, se observa en la figura 5.10 como el manipulador sigue la trayectoria ya establecida.

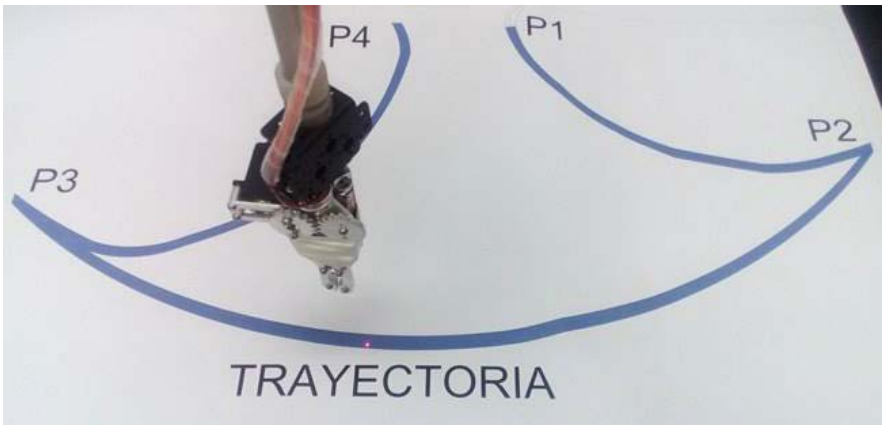


Figura 5. 10: Trayectoria mediante corrección de errores.

5.4 Conclusiones

Como se pudo observar el sistema implementado resultó ser adecuado y preciso para seguir una trayectoria propuesta, además corregir errores que se presenten en la transmisión de datos, con la ventaja de ser controlado en tiempo real.

CONCLUSIONES

- Para la transmisión de datos se realizó un análisis acerca de los protocolos de comunicación industriales, tomando en cuenta la disponibilidad de información de los dispositivos de campo, precisión y rango de datos, llegando a la conclusión de que el protocolo Modbus presenta mejores características para la transmisión de información entre maestro y esclavos, además de tener la ventaja de ser un *software* de código abierto.

- Para analizar el funcionamiento del protocolo ModBus que se implementó en código Python, fue necesario realizar tres pruebas: la primera mediante una conexión por cable utilizando el estándar RS232, la segunda de forma inalámbrica mediante módulos RF y por último utilizando módulos Xbee, como conclusión se llegó a determinar que el mejor medio de comunicación es de forma inalámbrica utilizando los módulos Xbee, debido a que son ideales para conexiones punto a punto, operan a una banda libre de 2.4 GHz y proveen conexiones seguras entre dispositivos haciéndolos ideales para la aplicación del proyecto, el alcance máximo logrado fue de 70 metros aproximadamente con línea de vista.

- Es importante concluir que la implementación del protocolo Modbus, se desarrolló un sistema tolerante a fallas basado en el código Hamming, para lo cual fue necesario realizar un codificador que es el encargado de agregar códigos de redundancia y un decodificador que tiene la función de corregir los errores de transmisión en el caso de que sea necesario. Para analizar el funcionamiento se implementó el codificador y decodificador en tarjetas PcDuino, donde se guardaron datos con y sin error para verificar el correcto funcionamiento del decodificador, es importante mencionar que además se realizaron pruebas de funcionamiento de forma virtual utilizando simuladores como: Modbus Poll, Modbus Slave, ModBus RTU y Spyder propia de la consola Python.

- El software que se utilizará para el desarrollo de este trabajo es Blender que es *open source*, se puede modelar en 3D, es una herramienta de fácil acceso para la creación de la animación del brazo robótico, además este programa tiene la ventaja de utilizar *scripts* de Python, lo cual facilitó la conexión entre el sistema SCADA creado y el programa maestro.

- Para analizar el correcto funcionamiento del sistema, se realizó una prueba de seguimiento de trayectoria por parte del manipulador de 3 GDL, para cual se aplicaron datos con y sin errores, obteniendo un correcto seguimiento de la trayectoria por parte del robot, a pesar de que los datos que llegaron al decodificador tenían errores.
- En base a lo descrito anteriormente el sistema de comunicación inalámbrica, el protocolo ModBus, el código corrector de errores Hamming y el seguimiento de la trayectoria por parte del manipulador, funcionaron correctamente.

RECOMENDACIONES

- Es necesario instalar las últimas actualizaciones de las librerías de ModBus en Python para tener un correcto funcionamiento y que los programas no tengan problemas en su vinculación.

- Para tener una óptima comunicación inalámbrica entre los Xbee se considera tener presente las características propias de estos módulos.

- Se recomienda realizar investigaciones con otros sistemas tolerantes a fallas, debido a que el código Hamming solamente realiza la corrección de un solo bit.

- Es importante analizar el comportamiento del sistema estudiado en esta tesis, para otros medios de comunicación inalámbrica.

- Luego del desarrollo de esta tesis es importante manifestar la importancia que tiene el lenguaje Python y por ende recomendar la utilización del mismo en diferentes proyectos de investigación.

BIBLIOGRAFÍA

Adafruit. (2015). *Adafruit*. Retrieved from https://www.adafruit.com/products/1063

Araujo, F. I. (2013). Diseño, contruccion e implementacion de un sistema para el manejo de un periferico de los acensores del edificio mediante comunicacion serial y protocolo modbus. *Politecnica nacional, I*, 134.

ASinterface. (2017, Febrero 02). Retrieved from http://www.as-interface.net/

Bailey, D., & Wright, E. (2013). *Practical SCADA for Industry*. Amsterdam: ELSEVIER.

Becerra, A. B. (2002). *Redes Industriales, Modbus y Ethernet Implementados en Automatas Programables Trilogi, Koyo y Modicon-Telemecanique.* 2002: Universidad industrial de Santander.

Berg, J. (2016). *minimalmodbus*. Retrieved Enero 14, 2017, from https://minimalmodbus.readthedocs.io/en/master/readme.html#features

CANopen. (2017, Febrero 10). *CAN in Automation*. Retrieved from https://www.can-cia.org/

Cebrián, R. P. (2012). *Una nueva Familia de Topologñias Híbridas para la Interconexión de Redes de Gran Escala.* Valencia: Universidad Politécnica de Valencia.

Comind. (2016). Comunicaciones Industriales. *Universidad de Oviedo*, 33.

Daneels, A., & Geneva. (1999). What is SCADA? *CDS*, 5.

Escuela de educaión técnica numero 2 lanus. (2007). Retrieved from Aplicaciones especificas: http://eet2lanus.tripod.com/R_1.htm

Foundation, F. (2015, Enero 1). *fieldbus*. Retrieved from http://www.fieldbus.org/

Gelvez, J. a. (2002). *Redes de comunicacion Industrial.* Santander: Universidad Industrial de Santander.

GitHub. (2017). *minimalmodbus*. Retrieved Enero 12, 2017, from https://github.com/ljean/modbus-tk

GitHub. (2017). *modbustk*. Retrieved Enero 12, 2017, from https://github.com/bashwork/pymodbus

Hussein, M., Jakllari, G., & Paillasa, B. (2015). Frugal Topologies for Saving Energy in IP Networks. *IEEE*, 9.

Interbus Organization. (n.d.). Retrieved from http://www.interbusclub.com/).

LinkSprite. (2016, Abril 14). *LinkSprite*. Retrieved from eCommerce: http://www.linksprite.com/

Lontorfos, G., Fairbanks, K. D., Watkins, L., & Robinson, W. H. (2015). Remotely Interferring Device Manipulation of Industrial Control Systems via Network Behavior. *IEEE*, 8.

lonworks.es. (2015, Marzo 18). Retrieved from http://www.lonworks.es/

Malavolta. (2007). *Escuela de educaión técnica numero 2 lanus*. Retrieved from Aplicaciones especificas: http://eet2lanus.tripod.com/R_1.htm

MODICON. (1996). *PI-MBUS_300 Rev.J.* Retrieved Diciembra 10, 2015, from Inc., Industrial Automation System: http://modbus.org/docs/PI_MBUS_300.pdf

Muñoz, A. R. (2009, Octubre 14). *Sistemas Industriales Distribuidos.* Retrieved from Universidad de Valencia: http://www.uv.es/rosado/courses/sid/Capitulo1_rev1.pdf

Networks, H. I. (2015). *anybus*. Retrieved from http://www.anybus.fr/products/controlnet.shtml

Olaya, A. F., Barandica Lopez, A., & Guerrero Moreno, F. (2004). Implementacion de una Red MODBUS/TCP. *Ingenieria y Competividad*, *6*(2), 11.

Organization, M. (2017). *MOdbus*. Retrieved from MOdbus Organization: http://www.modbus.org/

Penin, A. R. (2007). *Sistemas SCADA.* México D.F: MARCOMBO, .

Penin, A. R. (2012). Sistemas SCADA. Madrid: marcombo.

Penin, A. R. (2013). In *Sistemas SCADA.* Barcelona: marcombo.

Petroni, A. N. (2003). Redes de Comunicaciones. *IEEE, 1*(cap3), 36.

Polverini, F. (2017, Enero 04). *fpcontrol.* Retrieved Febrero 16, 2017, from Sistema de control centralizado (SCIC): http://www.fpcontrol.com.ar/Notas_SCIC.html

Profibus. (2016). *Profibus*. Retrieved Febrero 18, 2016, from Profinet: http://www.profibus.com/).

Pypi.python.org. (2017, Septiembre 10). *pyserial 3.2.1*. Retrieved from Python Package Index: https://pypi.python.org/pypi/pyserial

Robomart. (2015). *Robomart*. Retrieved ENERO 10, 2017, from https://www.robomart.com/max232-dual-eia-232-drivers-and-receiver

Rondon, O. M., & Arenas, J. I. (2011). *Redes de comunicaión industrial.* Bucaramanga: Universidad Industrial de Santander.

Shah, N. (2017). Auto Configuration of ACL Policy in case of Topology Change in Hybrid SDN. *IEEE*, 14.

SMART. (2017). Retrieved from Equipamentos Industriales: http://www.smar.com/espanol/devicenet

sparkfun. (2017, Enero 10). *SparkFun Electronics*. Retrieved from https://www.sparkfun.com/products/retired/10414

TECHONOLOGY, S. H. (2008, enero 14). *HAC-Smart Series RF Module*. Retrieved from User ’s Manual: http://www.rf-module-china.com/es/pdf/20090603181334qPPgJL.pdf

Torres, H. M. (2014). *Diseño e Implementación de un Sistema de Comunicación Industrial Tolerante a Fallas Basado en el Protocolo ModBus Aplicado al Control de un Robot Manipulador de 3 GDL*. Santiago: Universidad de Santiago de Chile.

Transced. (2017). Retrieved from https://es.transcend-info.com/Products/No-96

Trendnet. (n.d.). *TRENDnet* . Retrieved Enero 10, 2017, from http://www.trendnet.com/langsp/products/USB-adapters/TU-S9

worldfip. (2017). Retrieved from http://www.worldfip.org/

ANEXOS

Anexo 1: Codificador Hamming.

Nombre del archivo: codificador.py

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python

import numpy, random, serial, os, sys
import numpy as np
import modbus_tk
import modbus_tk.defines as cst
from modbus_tk import modbus_rtu

# Método para crear de forma aleatoria los mensajes de 4 bits
def generarNumeros():
    msg = []
    length = 4
    for i in range(length):
        letter = random.choice([0,1])
        msg.append(letter)
    return numpy.array(msg, dtype = int)

# Método para codificar el mensaje utilizando la multiplicación de matrices
# Multiplica el mensaje(m)por la matriz de codificación de Hamming (g)


def encode(m, g):

    enc = numpy.dot(m, g)%2

    return enc


# Método para crea de forma aleatoria el error
# Parte este prosedimiento copiando el mensaje original
# Si el valor aleatorio es menor al error fijado cambia el bit de la posición (i)
def noise(m, error, bit):
    noisy = numpy.copy(m)
    cont = 0
    for i in range(len(noisy)):
        e = random.random()
        if e <= error:
            if noisy[i] == 0:
                noisy[i] = 1
                cont +=1
            else:
                noisy[i] = 0
                cont +=1
            if cont == bit:
                break
    return noisy

#Codigo principal
def main():

    try:
        #Puerto para escribir datos en el esclavo
        master1 = modbus_rtu.RtuMaster(
            serial.Serial(port='COM1', baudrate=9600, bytesize=8, parity='N', stopbits=1, xonxoff=0)
        )
        master1.set_timeout(1)
        master1.set_verbose(True)

        g =  numpy.array([[1, 0, 0, 0, 0, 1, 1],[0, 1, 0, 0, 1, 0, 1],[0, 0, 1, 0, 1, 1, 0],[0, 0, 0, 1, 1, 1, 1]])
```

```
        cont = 0
        while (cont <= 9 ):
            master1.execute(1, cst.WRITE_SINGLE_REGISTER, 31, output_value = 1)
                                    #Llama a la funcion para generar numeros aleatorios de 4 bits
            msg = generarNumeros()
                                    #Transforma el numero binario a decimal
            y = 2**0*msg[3] + 2**1*msg[2] + 2**2*msg[1] + 2**3*msg[0]
            print "Dato original: ", y, msg
            master1.execute(1, cst.WRITE_SINGLE_REGISTER, cont, output_value = y)

                                    #Llama a la función para codificar el mensaje
            enc = encode(msg, g)
                                    #Transforma el numero binario a decimal
            c = 2**0*enc[6] + 2**1*enc[5] + 2**2*enc[4] + 2**3*enc[3] + 2**4*enc[2] + 2**5*enc[1] + 2**6*enc[0]
            print "Dato codificado sin error: ", c, enc
            master1.execute(1, cst.WRITE_SINGLE_REGISTER, cont + 10 , output_value = c)

                                    #Llama a la función para introducir el error en el mensaje
            noisy = noise(enc, 0.1, 1)
            #Transforma el numero binario a decimal
            d = 2**0*noisy[6] + 2**1*noisy[5] + 2**2*noisy[4] + 2**3*noisy[3] + 2**4*noisy[2] + 2**5*noisy[1] +
2**6*noisy[0]
            print "Dato codificado con error: ", d, noisy
            master1.execute(1, cst.WRITE_SINGLE_REGISTER, cont + 20 , output_value = d)
            print
            cont += 1
        master1.execute(1, cst.WRITE_SINGLE_REGISTER, 31, output_value = 0)
    except modbus_tk.modbus.ModbusError as exc:
        logger.error("%s- Code=%d", exc, exc.get_exception_code())


if __name__ == "__main__":
    main()
```

Anexo 2: Decodificador Hamming.

Nombre del archivo: decodificador.py

```
# -*- coding: utf-8 -*-

import serial, os, numpy, random, sys, time
import numpy as np
import modbus_tk
import modbus_tk.defines as cst
import modbus_tk.modbus_rtu as modbus_rtu
from modbus_tk import modbus_rtu

logger = modbus_tk.utils.create_logger(name="console", record_format="%(message)s")

if __name__ == "__main__":
    #Crea el esclavo.
    port = serial.Serial('COM2', 9600, parity = 'N', bytesize = 8, stopbits = 1, timeout = 0.01)
    server = modbus_rtu.RtuServer(port)
    server.start()
    slave_1 = server.add_slave(1)
    slave_1.add_block('0', cst.HOLDING_REGISTERS, 0, 33)

    # Método para decodificar el mensaje utilizando la multiplicación de matrices
    # Multiplica la matriz de decodificación de Hamming (h)por el mensaje
    def decode(port, h):
        dec = np.dot(h, port)%2
        return dec

    # Método para determinar la posición del error
    # Calcula el valor de la posición donde se encuentra el error en forma decimal
    # La posición que se considera es desde [0,1,2.....6]
    def findError(dec):
        n = ''
        for i in range(len(dec)):
            n += str(dec[i])
        n = (int(n,2))-1
        return n

    # Método para realiza la corrección del error
    # Si es [0] lo cambia a [1]
    # Si es [1] lo cambia a [0]
    def correct(port, n):
        if port[n] == 0:
            port[n] = 1
        else:
            port[n] = 0

        return port

    try:

        logger.info("Inicia Decodificador")
        print
        cont = 0
        while True:
            registros = slave_1.get_values("0", 0, 33)
            if (registros[29] > 0):
                registros = slave_1.get_values("0", 0, 33)
                while (cont <= 9):
                    #Transforma el valor decimal a binario de 8 bits del primer registro
                    a = np.array([[registros[cont + 20]]], dtype=np.uint8)
                    b = np.unpackbits(a)

                    #Pasa de 8 bits a 7 bits
                    lectura = np.delete(b, 0)
```

```
            h = np.array([[0, 0, 0, 1, 1, 1, 1],[0, 1, 1, 0, 0, 1, 1],[1, 0, 1, 0, 1, 0, 1]])
                        #Decodifica el valor de 7 bits
            dec = decode(lectura, h)
            d = 2**0*lectura[6] + 2**1*lectura[5] + 2**2*lectura[4] + 2**3*lectura[3] + 2**4*lectura[2] +
2**5*lectura[1] + 2**6*lectura[0]
            print "Dato codificado con error: ", d, lectura
            #Encuentra el error
            num = findError(dec)

                        #Corrige el error
            lectura = correct(lectura, num)
            c = 2**0*lectura[6] + 2**1*lectura[5] + 2**2*lectura[4] + 2**3*lectura[3] + 2**4*lectura[2] +
2**5*lectura[1] + 2**6*lectura[0]
            print "Dato decodificado sin error: ", c, lectura
            #Elimina los bits que ya no se necesitan
            lectura = np.delete(lectura, 6)
            lectura = np.delete(lectura, 5)
            lectura = np.delete(lectura, 4)
            cont += 1
            y = 2**0*lectura[3] + 2**1*lectura[2] + 2**2*lectura[1] + 2**3*lectura[0]

            print "Dato original: ", y, lectura


            print


    finally:

        server.stop()
```

Anexo 3: Código Python en el SCADA.

Nombre del archivo: interfaz.py

```
# -*- coding: utf-8 -*-

import serial, os, numpy, random, sys

import numpy as np

import pyqtgraph as pg

import modbus_tk

import time

import modbus_tk.defines as cst

from modbus_tk import modbus_rtu

from PyQt4 import uic, QtCore, QtGui


Qt = QtCore.Qt


try:

    _encoding = QtGui.QApplication.UnicodeUTF8

    def _translate(context, text, disambig):

        return QtGui.QApplication.translate(context, text, disambig, _encoding)

except AttributeError:

    def _translate(context, text, disambig):

        return QtGui.QApplication.translate(context, text, disambig)


class MyWindow(QtGui.QMainWindow):

    def __init__(self):

        super(MyWindow, self).__init__()

        uic.loadUi('Principal.ui', self)

        #Listar los puertos disponibles y cargarlos en los combobox

        self.combo1.addItems(self.cargarPuertos())
```

```
self.combo2.addItems(self.cargarPuertos())
self.combo3.addItem( 'Dato Original' )
self.combo3.addItem('Dato Codificado sin error (1)')
self.combo3.addItem('Dato Codificado sin error (2)')
self.combo3.addItem('Dato Codificado con error (1)')
self.combo3.addItem('Dato Codificado con error (2)')
self.combo4.addItem('Dato Original')
self.combo4.addItem('Dato Codificado sin error (1)')
self.combo4.addItem('Dato Codificado sin error (2)')
self.combo4.addItem('Dato Codificado con error (1)')
self.combo4.addItem('Dato Codificado con error (2)')
self.combo5.addItem('Dato Original')
self.combo5.addItem('Dato Codificado sin error (1)')
self.combo5.addItem('Dato Codificado sin error (2)')
self.combo5.addItem('Dato Codificado con error (1)')
self.combo5.addItem('Dato Codificado con error (2)')
#Declarar el timer
self.ctimer = QtCore.QTimer()
self.gtimer = QtCore.QTimer()
self.leer = [0, 0, 0]
#Variable para escribir en el esclavo
self.escribir = [0,0,0,0,0,0]
#Variables para habilitar el uso del timer
self.habilitar1 = False
self.habilitar2 = False
self.habilitar3 = False
self.desconectar.setEnabled(False)


#Variables para graficar
```

```
self.m1y = 0

self.m2y = 0

self.m3y = 0

self.m4y = 0

self.m5y = 0

self.m6y = 0

self.serror = [0, 0, 0, 0, 0, 0]

self.error = [0, 0, 0, 0, 0, 0]

self.codsine = [' ', ' ', ' ', ' ', ' ', ' ']

self.codcone = [' ', ' ', ' ', ' ', ' ', ' ']


# Plot in chunks, adding one new plot curve for every 100 samples

self.chunkSize = 100

# Remove chunks after we have 10

self.maxChunks = 10

self.startTime = pg.ptime.time()

self.curves = []

self.curves1 = []

self.curves2 = []

self.curves3 = []

self.curves4 = []

self.curves5 = []


self.data = np.empty((self.chunkSize+1,2))

self.data1 = np.empty((self.chunkSize+1,2))

self.data2 = np.empty((self.chunkSize+1,2))

self.data3 = np.empty((self.chunkSize+1,2))

self.data4 = np.empty((self.chunkSize+1,2))

self.data5 = np.empty((self.chunkSize+1,2))
```

```
self.ptr5 = 0
self.tiempo = 0
#Propiedades de los plots
self.Plot1.setYRange(0, 180)
self.Plot1.setXRange(-10, 0)
self.Plot1.setLabel('bottom', 'Tiempo', 's')
self.Plot2.setYRange(0, 180)
self.Plot2.setXRange(-10, 0)
self.Plot2.setLabel('bottom', 'Tiempo', 's')
self.Plot3.setYRange(0, 180)
self.Plot3.setXRange(-10, 0)
self.Plot3.setLabel('bottom', 'Tiempo', 's')
self.Plot4.setYRange(0, 180)
self.Plot4.setXRange(-10, 0)
self.Plot4.setLabel('bottom', 'Tiempo', 's')
self.Plot5.setYRange(0, 180)
self.Plot5.setXRange(-10, 0)
self.Plot5.setLabel('bottom', 'Tiempo', 's')
self.Plot6.setYRange(0, 180)
self.Plot6.setXRange(-10, 0)
self.Plot6.setLabel('bottom', 'Tiempo', 's')
#Señales cuando ocurre el timeout del timer
self.connect(self.ctimer, QtCore.SIGNAL("timeout()"), self.timerLectura)
#self.connect(self.gtimer, QtCore.SIGNAL("timeout()"), self.timerGraficos)
#Eventos cuando se pulsan los botones Conectar
self.connect(self.conectar1, QtCore.SIGNAL('clicked()'), self.conectarPuerto1)
self.connect(self.conectar2, QtCore.SIGNAL("clicked()"), self.conectarPuerto2)
self.connect(self.puertos, QtCore.SIGNAL("clicked()"), self.actualizarPuertos)
```

```
    self.connect(self.desconectar, QtCore.SIGNAL("clicked()"), self.desconectarServer)
    #Mostrar la aplicación
    self.show()

def desconectarServer(self):
    self.server.stop()
    self.conectar1.setEnabled(True)
    self.conectar2.setEnabled(True)
    self.desconectar.setEnabled(False)
    self.label_8.setText("Desconectado")
    self.label_7.setText("Desconectado")
    #self.gtimer.stop()
    self.ctimer.stop()

def actualizarPuertos(self):
    self.combo1.clear()
    self.combo2.clear()
    self.combo1.addItems(self.cargarPuertos())
    self.combo2.addItems(self.cargarPuertos())

def timerLectura(self):
    self.codificacion()

def timerGraficos(self):

    recibidos = self.master1.execute(1, cst.READ_HOLDING_REGISTERS, 9, 3)
    now = pg.ptime.time()
    for c in self.curves:
        c.setPos(-(now-self.startTime), 0)
```

```
for c1 in self.curves1:
    c1.setPos(-(now-self.startTime), 0)

for c2 in self.curves2:
    c2.setPos(-(now-self.startTime), 0)

for c3 in self.curves3:
    c3.setPos(-(now-self.startTime), 0)

for c4 in self.curves4:
    c4.setPos(-(now-self.startTime), 0)

for c5 in self.curves5:
    c5.setPos(-(now-self.startTime), 0)

i = self.ptr5 % self.chunkSize

if i == 0:
    curve = self.Plot1.plot(pen='r')
    curve1 = self.Plot2.plot(pen='b')
    curve2 = self.Plot3.plot(pen='g')
    curve3 = self.Plot4.plot(pen='r')
    curve4 = self.Plot5.plot(pen='b')
    curve5 = self.Plot6.plot(pen='g')

    self.curves.append(curve)
    self.curves1.append(curve1)
    self.curves2.append(curve2)
```

```
self.curves3.append(curve3)

self.curves4.append(curve4)

self.curves5.append(curve5)


last = self.data[-1]

last1 = self.data1[-1]

last2 = self.data2[-1]

last3 = self.data3[-1]

last4 = self.data4[-1]

last5 = self.data5[-1]


self.data = np.empty((self.chunkSize+1,2))

self.data[0] = last

self.data1 = np.empty((self.chunkSize+1,2))

self.data1[0] = last1

self.data2 = np.empty((self.chunkSize+1,2))

self.data2[0] = last2

self.data3 = np.empty((self.chunkSize+1,2))

self.data3[0] = last3

self.data4 = np.empty((self.chunkSize+1,2))

self.data4[0] = last4

self.data5 = np.empty((self.chunkSize+1,2))

self.data5[0] = last5


while len(self.curves) > self.maxChunks:

    c = self.curves.pop(0)

    self.Plot1.removeItem(c)


while len(self.curves1) > self.maxChunks:
```

```
            c1 = self.curves1.pop(0)
            self.Plot2.removeItem(c1)

        while len(self.curves2) > self.maxChunks:
            c2 = self.curves2.pop(0)
            self.Plot3.removeItem(c2)

        while len(self.curves3) > self.maxChunks:
            c3 = self.curves3.pop(0)
            self.Plot4.removeItem(c3)

        while len(self.curves4) > self.maxChunks:
            c4 = self.curves4.pop(0)
            self.Plot5.removeItem(c4)

        while len(self.curves5) > self.maxChunks:
            c5 = self.curves5.pop(0)
            self.Plot6.removeItem(c5)


    else:
        curve = self.curves[-1]
        curve1 = self.curves1[-1]
        curve2 = self.curves2[-1]
        curve3 = self.curves3[-1]
        curve4 = self.curves4[-1]
        curve5 = self.curves5[-1]

    self.data[i+1,0] = now - self.startTime
```

```
if str(self.combo3.currentText()) == 'Dato Original':
    self.data[i+1,1] = self.m1y
    curve.setData(x=self.data[:i+2, 0], y=self.data[:i+2, 1])
elif str(self.combo3.currentText()) == 'Dato Codificado sin error (1)':
    self.data[i+1,1] = self.serror[0]
    curve.setData(x=self.data[:i+2, 0], y=self.data[:i+2, 1])
elif str(self.combo3.currentText()) == 'Dato Codificado sin error (2)':
    self.data[i+1,1] = self.serror[1]
    curve.setData(x=self.data[:i+2, 0], y=self.data[:i+2, 1])
elif str(self.combo3.currentText()) == 'Dato Codificado con error (1)':
    self.data[i+1,1] = self.error[0]
    curve.setData(x=self.data[:i+2, 0], y=self.data[:i+2, 1])
elif str(self.combo3.currentText()) == 'Dato Codificado con error (2)':
    self.data[i+1,1] = self.error[1]
    curve.setData(x=self.data[:i+2, 0], y=self.data[:i+2, 1])


self.data1[i+1,0] = now - self.startTime
if str(self.combo4.currentText()) == 'Dato Original':
    self.data1[i+1,1] = self.m2y
    curve1.setData(x=self.data1[:i+2, 0], y=self.data1[:i+2, 1])
elif str(self.combo4.currentText()) == 'Dato Codificado sin error (1)':
    self.data1[i+1,1] = self.serror[2]
    curve1.setData(x=self.data1[:i+2, 0], y=self.data1[:i+2, 1])
elif str(self.combo4.currentText()) == 'Dato Codificado sin error (2)':
    self.data1[i+1,1] = self.serror[3]
    curve1.setData(x=self.data1[:i+2, 0], y=self.data1[:i+2, 1])
elif str(self.combo4.currentText()) == 'Dato Codificado con error (1)':
    self.data1[i+1,1] = self.error[2]
    curve1.setData(x=self.data1[:i+2, 0], y=self.data1[:i+2, 1])
```

```
elif str(self.combo4.currentText()) == 'Dato Codificado con error (2)':
    self.data1[i+1,1] = self.error[3]
    curve1.setData(x=self.data1[:i+2, 0], y=self.data1[:i+2, 1])

self.data2[i+1,0] = now - self.startTime
if str(self.combo5.currentText()) == 'Dato Original':
    self.data2[i+1,1] = self.m3y
    curve2.setData(x=self.data2[:i+2, 0], y=self.data2[:i+2, 1])
elif str(self.combo5.currentText()) == 'Dato Codificado sin error (1)':
    self.data2[i+1,1] = self.serror[4]
    curve2.setData(x=self.data2[:i+2, 0], y=self.data2[:i+2, 1])
elif str(self.combo5.currentText()) == 'Dato Codificado sin error (2)':
    self.data2[i+1,1] = self.serror[5]
    curve2.setData(x=self.data2[:i+2, 0], y=self.data2[:i+2, 1])
elif str(self.combo5.currentText()) == 'Dato Codificado con error (1)':
    self.data2[i+1,1] = self.error[4]
    curve2.setData(x=self.data2[:i+2, 0], y=self.data2[:i+2, 1])
elif str(self.combo5.currentText()) == 'Dato Codificado con error (2)':
    self.data2[i+1,1] = self.error[5]
    curve2.setData(x=self.data2[:i+2, 0], y=self.data2[:i+2, 1])

self.data3[i+1,0] = now - self.startTime
self.data3[i+1,1] = recibidos[0]
curve3.setData(x=self.data3[:i+2, 0], y=self.data3[:i+2, 1])
self.recibido1.setText(str(recibidos[0]))

self.data4[i+1,0] = now - self.startTime
self.data4[i+1,1] = recibidos[1]
curve4.setData(x=self.data4[:i+2, 0], y=self.data4[:i+2, 1])
```

```
    self.recibido2.setText(str(recibidos[1]))


    self.data5[i+1,0] = now - self.startTime
    self.data5[i+1,1] = recibidos[2]
    curve5.setData(x=self.data5[:i+2, 0], y=self.data5[:i+2, 1])
    self.recibido3.setText(str(recibidos[2]))


    self.ptr5 += 1


def cargarPuertos(self):
    available = []
    if os.name == "nt":
        for i in range(256):
            try:
                s = serial.Serial('COM' + str(i))
                available.append( s.portstr)
                s.close()   # explicit close 'cause of delayed GC in java
            except serial.SerialException:
                pass


    return available


def conectarPuerto1(self):
    if len(self.cargarPuertos()) == 0:
            QtGui.QMessageBox.critical(None,
            self.trUtf8("Error"),
            self.trUtf8("""No existen puertos disponibles."""),
            QtGui.QMessageBox.StandardButtons(\
                QtGui.QMessageBox.Ok))
```

```
        else:
            self.interfaz = serial.Serial(str(self.combo1.currentText()), baudrate=9600, timeout=5)
            self.label_8.setText("Conectado a " + str(self.combo1.currentText()))
            self.server = modbus_rtu.RtuServer(self.interfaz)
            self.server.start()
            # Define esclavo, dirección y registros de mantenimiento a utilizar
            self.slave_1 = self.server.add_slave(1)
            self.slave_1.add_block('0', cst.HOLDING_REGISTERS, 0, 13)
            self.conectar1.setEnabled(False)
            self.desconectar.setEnabled(True)
            self.habilitar1 = True
            if (self.habilitar1 == True) and (self.habilitar2 == True):
                self.ctimer.start(0.1)
                #self.gtimer.start(10)


    def conectarPuerto2(self):
        if len(self.cargarPuertos()) == 0:
                QtGui.QMessageBox.critical(None,
                self.trUtf8("Error"),
                self.trUtf8("""No existen puertos disponibles."""),
                QtGui.QMessageBox.StandardButtons(\
                    QtGui.QMessageBox.Ok))
        else:
            self.master1 = modbus_rtu.RtuMaster(
                serial.Serial(port=str(self.combo2.currentText()), baudrate=9600, bytesize=8, parity='N', stopbits=1,
xonxoff=0)
            )
            self.master1.set_timeout(1)
            self.master1.set_verbose(True)
```

```
        self.label_7.setText("Conectado a " + str(self.combo2.currentText()))
        self.conectar2.setEnabled(False)
        self.habilitar2 = True
        self.desconectar.setEnabled(True)
        if (self.habilitar1 == True) and (self.habilitar2 == True):
            self.ctimer.start(0.1)
            #self.gtimer.start(10)

def codificacion(self):



    ##############################################################################
    # Método para codificar el mensaje utilizando la multiplicación de matrices
    # Multiplica el mensaje(m)por la matriz de codificación de Hamming (g)
    ##############################################################################
    def encode(m, g):
        enc = numpy.dot(m, g)%2
        return enc


    def split_list(a_list):
        half = len(a_list)/2
        return a_list[:half], a_list[half:]


    ##############################################################################
    # Método para crear de forma aleatoria el error
    # Parte este procedimiento copiando el mensaje original
    # Si el valor aleatorio es menor al error fijado cambia el bit de la posición (i)
    ##############################################################################
```

```
def noise(m, error, bit):
    noisy = numpy.copy(m)
    cont = 0
    for i in range(len(noisy)):
        e = random.random()
        if e <= error:
            if noisy[i] == 0:
                noisy[i] = 1
                cont +=1
            else:
                noisy[i] = 0
                cont +=1
            if cont == bit:
                break
    return noisy



g =  numpy.array([[1, 0, 0, 0, 0, 1, 1],[0, 1, 0, 0, 1, 0, 1],[0, 0, 1, 0, 1, 1, 0],[0, 0, 0, 1, 1, 1, 1]])


#Lectura de los registros
port = self.slave_1.get_values("0", 0, 13)


bin1 = np.array([[port[0]]], dtype=np.uint8)
bin1 = np.unpackbits(bin1)
bin2 = np.array([[port[1]]], dtype=np.uint8)
bin2 = np.unpackbits(bin2)
bin3 = np.array([[port[2]]], dtype=np.uint8)
bin3 = np.unpackbits(bin3)
msb1, lsb1 = split_list(bin1)
```

```
msb2, lsb2 = split_list(bin2)

msb3, lsb3 = split_list(bin3)

#Codifica el primer byte sin error

enc1 = encode(msb1, g)

enc2 = encode(lsb1, g)

#Codifica el segundo byte sin error

enc3 = encode(msb2, g)

enc4 = encode(lsb2, g)

#Codifica el tercer byte sin error

enc5 = encode(msb3, g)

enc6 = encode(lsb3, g)

                #Codifica el primer byte con error

err1 = noise(enc1, 0.1, 1)

err2 = noise(enc2, 0.1, 1)

                #Codifica el segundo byte con error

err3 = noise(enc3, 0.1, 1)

err4 = noise(enc4, 0.1, 1)

#Codifica el tercer byte con error

err5 = noise(enc5, 0.1, 1)

err6 = noise(enc6, 0.1, 1)



                #valores decimales

reg0 = 2**0*err1[6] + 2**1*err1[5] + 2**2*err1[4] + 2**3*err1[3] + 2**4*err1[2] + 2**5*err1[1] +
2**6*err1[0]

reg1 = 2**0*err2[6] + 2**1*err2[5] + 2**2*err2[4] + 2**3*err2[3] + 2**4*err2[2] + 2**5*err2[1] +
2**6*err2[0]

reg2 = 2**0*err3[6] + 2**1*err3[5] + 2**2*err3[4] + 2**3*err3[3] + 2**4*err3[2] + 2**5*err3[1] +
2**6*err3[0]

reg3 = 2**0*err4[6] + 2**1*err4[5] + 2**2*err4[4] + 2**3*err4[3] + 2**4*err4[2] + 2**5*err4[1] +
2**6*err4[0]

reg4 = 2**0*err5[6] + 2**1*err5[5] + 2**2*err5[4] + 2**3*err5[3] + 2**4*err5[2] + 2**5*err5[1] +
2**6*err5[0]
```

```
    reg5 = 2**0*err6[6] + 2**1*err6[5] + 2**2*err6[4] + 2**3*err6[3] + 2**4*err6[2] + 2**5*err6[1] +
2**6*err6[0]


    #Escribe los datos codificados en los registros del esclavo

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 0, output_value=reg0)

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 1, output_value=reg1)

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 2, output_value=reg2)

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 3, output_value=reg3)

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 4, output_value=reg4)

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 5, output_value=reg5)

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 6, output_value=port[0])

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 7, output_value=port[1])

    self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 8, output_value=port[2])


    if port[3] == 1:

       self.master1.execute(1, cst.WRITE_SINGLE_REGISTER, 12, output_value=1)


    self.m1y = port[0]

    self.m2y = port[1]

    self.m3y = port[2]

    self.etq1.setText(str(port[0]))

    self.etq2.setText(str(port[1]))

    self.etq3.setText(str(port[2]))


    self.label_21.setText(str(port[0]))

    self.label_19.setText(str(port[1]))

    self.label_18.setText(str(port[2]))


    self.label_23.setText(str(enc1))
```

```
            self.label_24.setText(str(enc2))
            self.label_26.setText(str(enc3))
            self.label_25.setText(str(enc4))
            self.label_30.setText(str(enc5))
            self.label_29.setText(str(enc6))

            self.label_35.setText(str(err1))
            self.label_32.setText(str(err2))
            self.label_38.setText(str(err3))
            self.label_33.setText(str(err4))
            self.label_31.setText(str(err5))
            self.label_34.setText(str(err6))
            #time.sleep(0.01)
            self.timerGraficos()



            #except modbus_tk.modbus.ModbusError as exc:
            #    logger.error("%s- Code=%d", exc, exc.get_exception_code())



if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    sys.exit(app.exec_())
```