Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño y desarrollo de una red MODBUS RTU basada en Arduino

Autor: Juan José Rabadán Barastegui
Tutor: Ángel Rodríguez Castaño

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Proyecto Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Diseño y desarrollo de una red MODBUS RTU basada en Arduino

Autor:
Juan José Rabadán Barastegui

Tutor:
Ángel Rodríguez Castaño
Profesor

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Grado: Diseño y desarrollo de una red MODBUS RTU basada en Arduino

Autor: Juan José Rabadán Barastegui

Tutor: Ángel Rodríguez Castaño

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

A mis amigos

Agradecimientos

A mi familia y amigos, sin cuyo apoyo nada de esto habría sido posible.

Juan José Rabadán Barastegui

Sevilla, 2017

Resumen

Hoy en día existen infinidad de sensores y actuadores para monitorizar y controlar todo tipo de parámetros físicos. Esta gran diversidad de dispositivos es en sí misma un hándicap a la hora de integrarlos en un mismo proyecto, pues surge la necesidad de crear un protocolo de comunicación común a todos ellos.

En este trabajo se ha usado MODBUS; un protocolo de comunicación muy extendido en dispositivos de automatización industrial y que está jugando un papel muy importante también en domótica dada su sencillez y fiabilidad.

El objeto final del proyecto es montar una red de comunicación maestro-esclavos usando como dispositivos inteligentes placas Arduino que puedan intercambiar información sobre determinados sensores y actuadores usando para comunicarse el protocolo MODBUS.

Abstract

Nowadays there are a vast number of sensors and actuators to monitor and control all types of physical parameters. This great diversity is itself a handicap when it comes to integrating them in the same project, as the need arises to create a communication protocol common to all of them.

MODBUS has been used in this project; a communication protocol widely used in industrial automation devices, which is playing a very important role in home automation too due to its simplicity and reliability.

The final aim of this project is to set up a master-slaves communications network using intelligent devices such as Arduino boards that can exchange information about certain sensors and actuators using the MODBUS protocol.

Índice

Índice de Tablas

Índice de Figuras

1 INTRODUCCIÓN

Cualquier tecnología suficientemente avanzada es indistinguible de la magia.

- Arthur C. Clarke -

Desde que Modicon creó MODBUS en 1979 para sus *PLCs* no ha dejado de extenderse su uso haciéndose estándar de facto por tener las siguientes características:

- Es público y gratuito
- Es fácil de implementar y no requiere mucho desarrollo
- Maneja bloques de datos sin restricciones

Por todo lo anterior se considera el protocolo adecuado para cubrir las necesidades requeridas para este trabajo y se analizará en profundidad más adelante.

Por otra parte Arduino es una compañía de hardware libre que ha creado placas de desarrollo basadas en microcontroladores a un precio muy reducido. Estas placas contienen pines digitales y analógicos de entrada y salida y generalmente un puerto USB para poder programarla con un computador haciendo uso de su entorno de desarrollo (*IDE),* que también es gratuito y de código abierto.

Por todo ello los nodos inteligentes serán placas Arduino que se comunicarán usando el protocolo MODBUS; los esclavos se conectarán a los sensores y serán los encargados de recolectar la información y enviarla al nodo maestro que será el encargado de la toma de decisiones gestionando los actuadores en función de la información recibida, como puede ser activar un led de aviso cuando un determinado parámetro sobrepase el límite establecido.

1.1 Antecedentes

Este trabajo es la continuación de un proyecto de curso de la asignatura Redes Industriales, optativa del último curso del Grado en Ingeniería de las Tecnologías de Telecomunicación. El objetivo de ese trabajo era lograr la comunicación maestro-esclavo entre dos nodos Arduino. Una vez se consiguió dicho objetivo se pensó la posibilidad de ampliarlo y montar una red MODBUS con un maestro y varios esclavos como posible Trabajo de Fin de Grado, dando lugar a este texto.

1.2 Metodología

Para alcanzar el objetivo se ha realizado un estudio de:

- Módulos comerciales compatibles con Arduino que soporten el protocolo MODBUS
- Sensores y actuadores compatibles con Arduino
- Instalación del *IDE* de Arduino en sistema operativo *Windows 7*
- Instalación de librerías de los Módulos MODBUS, sensores y actuadores
- Conexión física más adecuada entre las placas Arduino, sensores y actuadores
- Programación del maestro y los esclavos

2 TECNOLOGÍA EMPLEADA

En este capítulo se dará una explicación teórica de las diferentes tecnologías que usarán los componentes del sistema.

2.1 Diseño de la aplicación y tecnologías empleadas

Para montar la red MODBUS RTU se van a usar tres placas Arduino, uno de ellos actuará como maestro y los restantes serán los esclavos. Comenzando por la capa física para conectar los Arduino se van a usar pares trenzados de cables para conexionado de placas de prueba ya que al no requerir abarcar grandes distancias son los más económicos. Para la transmisión física se usará el estándar RS-485 que se detallará posteriormente.

A los dos esclavos se conectarán diferentes sensores, como podrían ser el sensor de movimiento (*PIR)* o el de humedad. Estos sensores enviarán la información a los esclavos que guardarán la información en sus registros MODBUS. Información que será leída periódicamente por parte del maestro consiguiendo así centralizar la monitorización en el dispositivo maestro tal como hacen los sistemas *SCADA* en entornos industriales. Es ya tarea del maestro decidir qué hacer en base a la información leída de los sensores. Pudiendo encender desde un led o una alarma cuando el sensor *PIR* detecta presencia hasta un ventilador cuando el sensor de temperatura sobrepase un límite. Estos ejemplos se usarán para demostrar el funcionamiento de la red, quedando la aplicación final totalmente abierta a la imaginación, puesto que según los sensores y actuadores que se usen la funcionalidad será completamente distinta.

2.2 Transmisión física

La transmisión física constituye la forma en que los bits se transforman en señales adecuadas para ser transmitidas por el medio físico también conocido como canal. Dependiendo de cuál sea el medio, el canal tendrá determinadas propiedades que hacen que haya señales que sean más o menos robustas a la atenuación del canal.

2.2.1 RS-485

RS-485 o EIA-485 se hizo estándar en 1983 por la *TIA/EIA*. Es un estándar de comunicaciones en bus para la capa física del modelo *OSI* y define las características eléctricas de los transmisores y receptores. La transmisión es serial y asíncrona, lo cual quiere decir que los bits se van transmitiendo uno detrás de otro y sin una señal que sincronice transmisor y receptor. El medio físico es un par trenzado (A, B) que admite hasta 256 estaciones. La comunicación es semidúplex y se pueden cubrir hasta 1200 metros con un mismo bus sin pérdida de información gracias a la transmisión diferencial que cancela gran cantidad de ruido. Las velocidades de transmisión oscilan entre los 300 y los 19200 bit/s.

Las especificaciones del estándar se resumen en la siguiente tabla:

RS-485	
Estándar	TIA/EIA-485-A
Medio físico	Par trenzado
Topología de red	Punto a punto, punto a multipunto, multi-drop
Modo de comunicación	Semiduplex, dúplex
Máximo de dispositivos	Originalmente 32, actualmente 256 e incluso más usando repetidores
Modo de operación	Diferencial
Niveles de tensión	-7V / +12V
“1” Lógico	Tensión positiva (B-A > +200mV)
“0” Lógico	Tensión negativa (B-A < -200mV)

Tabla 2-1: Especificaciones del estándar RS-485

La transmisión diferencial se consigue transmitiendo en cada cable la misma señal desfasada 180 grados, de manera que el ruido afecta por igual ambos cables e invirtiendo una de las señales se consigue que el ruido se cancele al sumarlas como se puede ver en la siguiente figura:

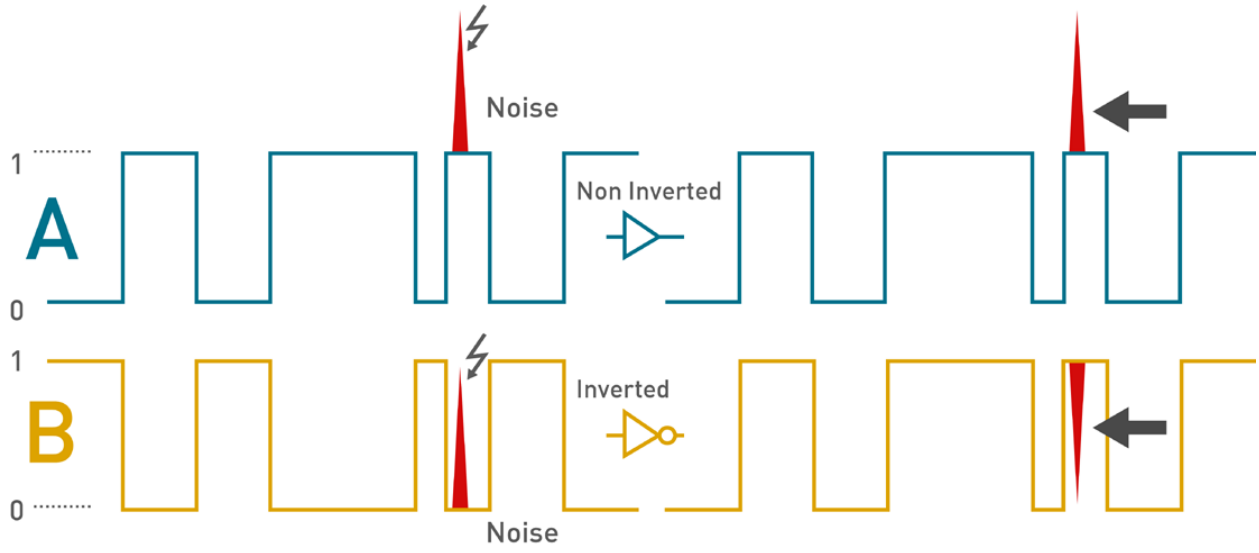

Figura 2-1: Transmisión diferencial

El esquema de la topología de red se puede ver en la siguiente figura:

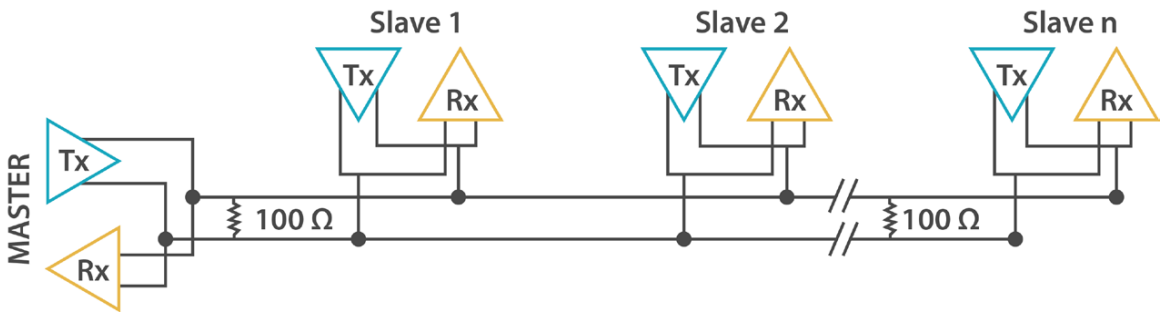

Figura 2-2: Topología de Red RS-485

Como se puede observar hacen falta sendas resistencias de terminación para que no se produzcan reflexiones en el bus. El valor de estas resistencias depende de la impedancia característica del cable y deberá ser calculado apropiadamente o consultado con el fabricante. El número máximo de dispositivos que se pueden conectar a la red es de 32, pudiéndose aumentar a 256 funcionando con entrada en alta impedancia e incluso a miles si se usan también regeneradores de señal.

2.3 El protocolo MODBUS

En este apartado se hará una descripción del protocolo MODBUS sobre línea serie basada en las especificaciones que se pueden descargar de www.modbus.org.

El protocolo MODBUS es una especificación que define un protocolo de nivel de aplicación que, aunque no cumple todo el modelo *OSI*, se podría enmarcar en los niveles 2 y 7 para MODBUS *serie* y MODBUS *TCP/IP* respectivamente. MODBUS no especifica cuál ha de ser la capa física, aunque la más usada es el soporte metálico y su velocidad de transmisión va desde los 75 a los 19200 baudios recomendados por el estándar, aunque en condiciones adecuadas se pueden configurar velocidades mayores.

La topología del protocolo es maestro-esclavo, pudiendo existir uno o varios esclavos que responderán las peticiones del maestro, ya que sólo éste puede iniciar la comunicación. Teniendo en cuenta lo anterior se puede decir que los esclavos son por analogía servidores y el maestro actúa como cliente.

MODBUS serie tiene dos modos de funcionamiento que se detallarán posteriormente; *ASCII o RTU* dependiendo del formato en que se codifique la información dentro de la trama ya sea en ASCII o en binario.

2.3.1 Modelo de datos

Según su tamaño podemos clasificar en dos los tipos de datos que maneja el protocolo: bits individuales y registros de 2 Bytes. Los bits individuales para entradas y salidas digitales y los registros para variables que requieran mayor tamaño. En la siguiente tabla se muestran los tipos de datos disponibles:

Tipo de objeto	Acceso	Tamaño
Discrete input	Solo leer	1 bit
Coil	Leer/escribir	1 bit
Input register	Solo leer	16 bits
Holding register	Leer/escribir	16 bits

Tabla 2-2: Modelo de datos de MODBUS

- **Discrete input o entrada discreta:** Puede ser generado por un sistema de entrada/salida.
- **Coil o bobina:** Puede ser modificado por un programa de aplicación.
- **Input register o registro de entrada:** Puede ser generado por un sistema de entrada/salida.
- **Holding register o registro de salida:** Puede ser modificado por un programa de aplicación.

2.3.2 Modelo de direccionamiento

Las direcciones de memoria para acceder a los datos de un dispositivo MODBUS están ordenadas según los tipos de datos:

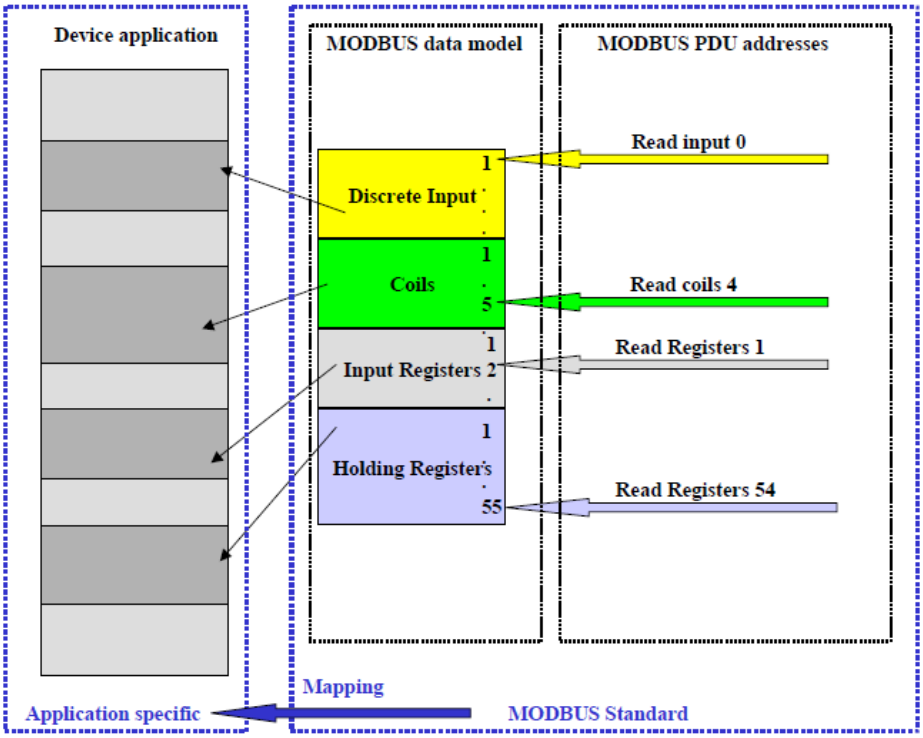

Figura 2-3: Modelo de direccionamiento

Tal como muestra la figura existen cuatro bloques que corresponden al modelo de datos. Todas las direcciones están referenciadas a cero de manera que si queremos leer el elemento N de un determinado bloque, éste será direccionado como N-1.

El mapeado de las direcciones MODBUS con las direcciones reales del dispositivo es independiente del estándar MODBUS, dependiendo totalmente del fabricante.

2.3.3 Formato de trama

El protocolo de aplicación MODBUS define una unidad de datos de protocolo (PDU) independientemente de las capas inferiores: pero en MODBUS serie hay que añadir algunos campos adicionales para construir una PDU adecuada para establecer la comunicación en redes o buses.

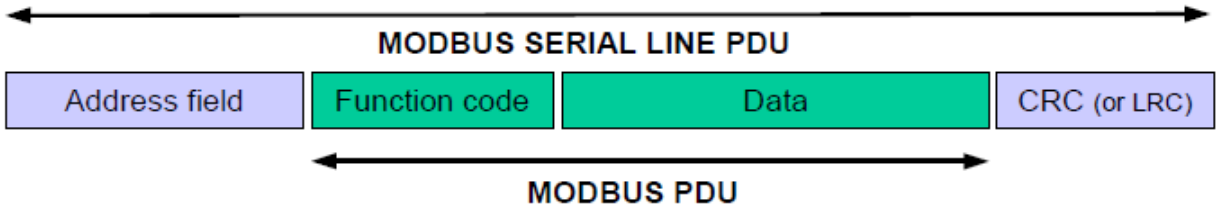

Figura 2-4: trama MODBUS serie

- **Address field o Campo de dirección:** Sirve para indicar la dirección del esclavo al que va dirigida la trama. El rango válido va desde 0 a 247, siendo el 0 la dirección de *Broadcast* y quedan reservadas las direcciones 248 a 255. Cuando el esclavo recibe una trama dirigida a él; construye la respuesta y pone su propia dirección en este campo, para que el maestro sepa de qué esclavo viene la respuesta.
- **Function code o Código de función:** Indica el código de la operación que el maestro solicita al esclavo; por ejemplo, leer un determinado registro.
- **Data o Campo de datos:** Lleva la información que se necesite para realizar determinada función; por ejemplo, escribir un valor en el registro indicado.
- **CRC o LRC:** Chequeo de redundancia cíclica o chequeo de redundancia longitudinal: sirve para asegurarse de que la información llega sin errores.

2.3.4 Diagrama de estado del maestro

El siguiente diagrama explica el comportamiento del maestro:

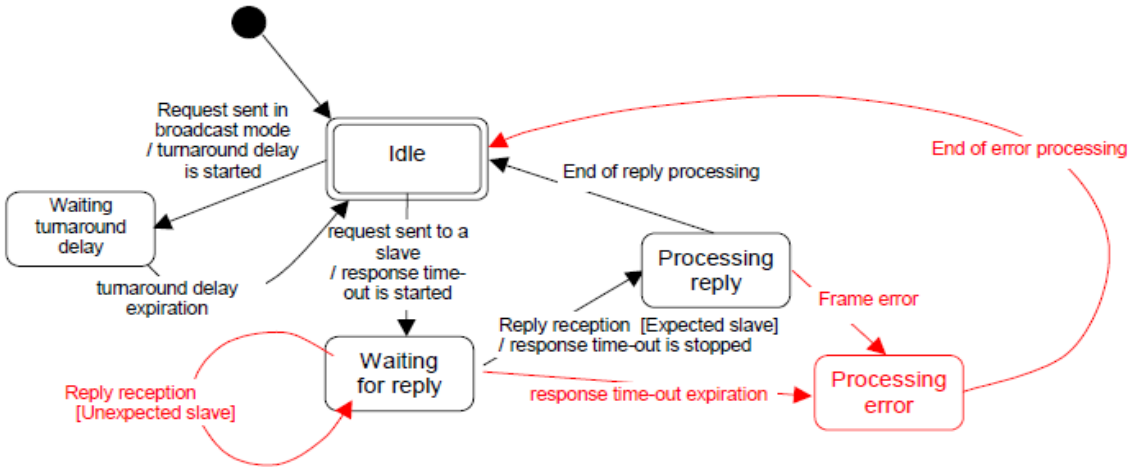

Figura 2-5: Diagrama de estados del maestro

- Existen 5 estados: "Idle" es el estado inicial de reposo, sólo se pueden enviar solicitudes desde este estado y no se sale de él a no ser que se envíe una solicitud en modo broadcast (dirigida a todos los esclavos) o unicast (dirigida a un único esclavo).
- Cuando se manda una solicitud en modo broadcast se activa una espera "turnaround" y permanece en estado de "Espera turnaround" para que los esclavos tengan tiempo de atender la solicitud antes de volver al estado de reposo desde el cuál se podría lanzar una solicitud nueva.
- Si se manda una solicitud en modo unicast se activa un temporizador y se pasa al estado "Esperando respuesta". De este estado se sale si: expira el temporizador; pasando al estado "Procesar error", o si se recibe una respuesta; deteniendo el temporizador y pasando al estado "Procesar respuesta".
- Por último tras procesar la respuesta vuelve al estado de reposo en caso de que la respuesta fuese correcta (no se detectaron errores de paridad ni de CRC/LRC) o pasa al estado "Procesar error" tras el cual vuelve de nuevo al estado de reposo desde el cuál se realizaría un reintento. El número de

reintentos depende de la configuración del maestro.

- Los errores de trama consisten en:
  - Comprobación de paridad de cada carácter.
  - Comprobación de redundancia de la trama completa.

2.3.5 Diagrama de estado del esclavo

El siguiente diagrama muestra el comportamiento del esclavo:

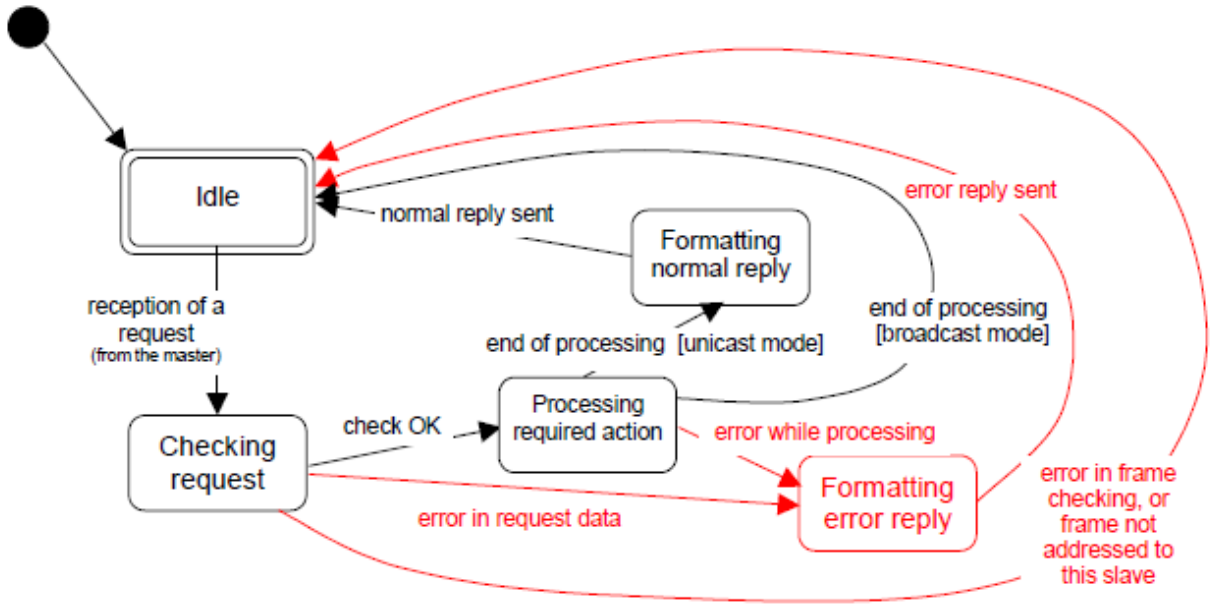

Figura 2-6: Diagrama de estados del esclavo

- "Idle" es el estado inicial. Si se recibe una solicitud del maestro se pasa al estado "Checking request" en el cual se comprueba si la solicitud no tiene errores. Si la solicitud es correcta se pasa al estado "Processing required action" y si es incorrecta hay dos posibilidades; que el error esté en los datos de la solicitud o que la trama sea errónea o no dirigida al esclavo que la recibe.
- Cuando el error está en los datos de la solicitud se pasa al estado "Formatting error reply" donde se monta una respuesta notificando el error y se envía al maestro pasando tras esto al estado de reposo "Idle".
- Cuando la trama es errónea o no está dirigida al esclavo que la recibe el esclavo la ignora y vuelve al estado de reposo.
- Desde el estado "Processing required action" hay tres formas de salir:
  - Pasando al estado "Formatting normal reply" si se completó la acción de la solicitud sin errores y ésta fue recibida en modo unicast.
  - Pasando al estado "Formatting error reply" si hubo algún error al realizar la acción solicitada.
  - Pasando directamente al estado de reposo si se completó correctamente la acción solicitada y ésta se recibió en modo broadcast.

2.3.6 Diagrama temporal de la comunicación maestro / esclavo

Este diagrama muestra tres posibles intercambios de información entre maestro y esclavo:

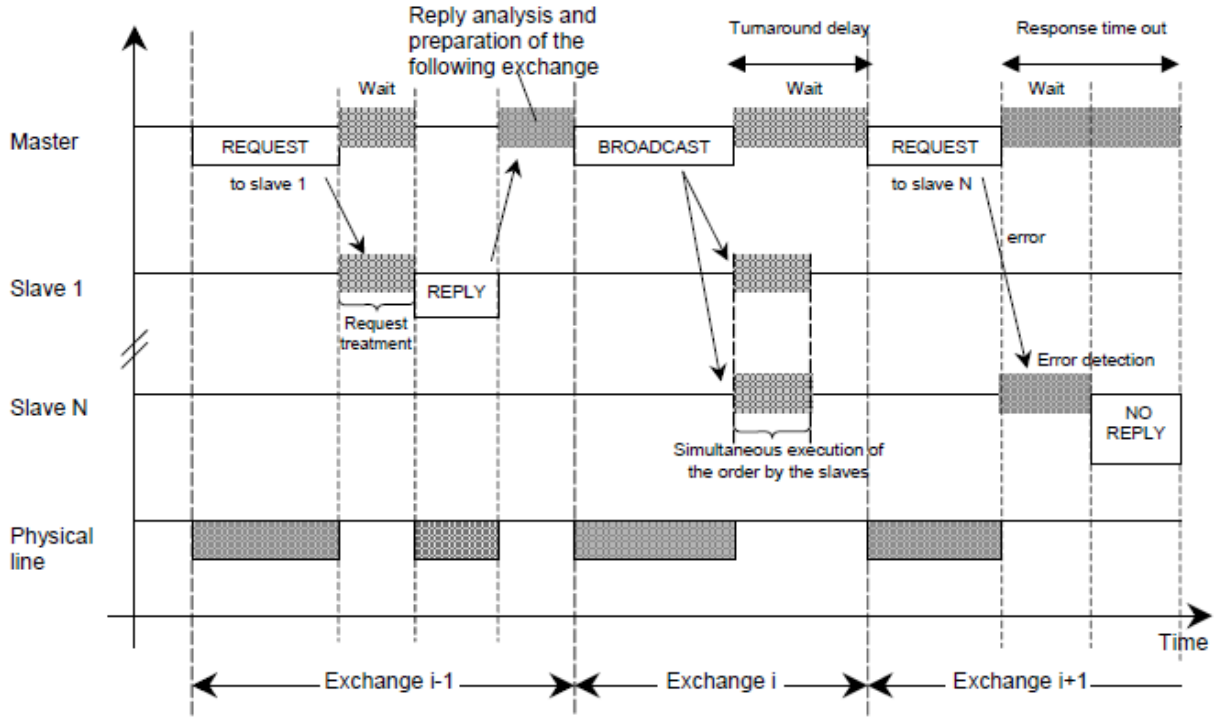

Figura 2-7: Diagrama temporal de la comunicación maestro / esclavo

- El primer intercambio (el primero empezando por la izquierda) es una solicitud en modo unicast que el esclavo 1 completa sin errores y envía la respuesta al maestro. Se pueden ver en gris los tiempos de espera del maestro y de procesamiento de la solicitud del esclavo.
- El segundo intercambio es una solicitud en modo broadcast. Se puede ver como los esclavos una vez la reciben la ejecutan en paralelo y como el maestro permanece a la espera un "Turnaround delay" antes de volver a enviar otra solicitud.
- El tercer y último intercambio es una solicitud al esclavo N; el esclavo detecta un error en la trama y la ignora, el maestro espera un tiempo hasta que expira el temporizador a partir de entonces, aunque no se ve reflejado en el diagrama, podría enviar un reintento.
- Los tiempos de Request y Broadcast dependen de las características de la trama (longitud, throughput)
- Los tiempos de espera y de tratamiento de la solicitud dependen de la capacidad del esclavo para el procesamiento de la solicitud.

2.3.7 Modo de transmisión *RTU*

En el modo de transmisión *RTU* para transmitir un Byte de información se necesitan agregar unos bits de inicio y stop para que el receptor sepa cuando empieza y acaba la información.

Como se puede ver en la figura existen dos modos dependiendo de si se comprueba la paridad o no. El modo definido por defecto por el estándar MODBUS es el de paridad par, quedando los modos de paridad impar o no paridad como opcionales.

El bit de paridad es un bit que se pone a "1" de manera que el número total de bits a "1" en el Byte que se está enviando coincida con el modo de paridad elegido.

Un ejemplo: se transmite "10010001" y el modo configurado es paridad par; como hay tres bits a "1", para que el número de bits total sea par el bit de paridad se tendrá que poner a "1". En caso contrario se pondría a "0".

Los bits se transmiten en el siguiente orden: de izquierda a derecha; desde el bit menos significativo (LSB) hasta el más significativo (MSB).

En el modo de no paridad como se puede observar se sustituye dicho bit por otro bit de stop.

Con chequeo de paridad

Inicio	1	2	3	4	5	6	7	8	Par	Stop

Sin chequeo de paridad

Inicio	1	2	3	4	5	6	7	8	Stop	Stop

Figura 2-8: Secuencia de Bits en modo RTU

Con todo lo anterior queda claro cómo se envían los Bytes de información de cada campo y conviene enfatizar que para transmitir un Byte de información en modo *RTU* se necesitan enviar 11 bits. La trama completa se puede ver en la siguiente figura:

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low, CRC Hi

Figura 2-9: Trama MODBUS RTU

Una vez montada la trama sólo falta saber cómo se envía. La siguiente figura muestra cómo se identifica el inicio y final de una trama; gracias a unos intervalos de silencio de al menos 3.5 veces el tiempo que se tarda en enviar un carácter que llamaremos a partir de ahora t3.5.

MODBUS message

Start	Address	Function	Data	CRC Check	End
≥ 3.5 char	8 bits	8 bits	N x 8 bits	16 bits	≥ 3.5 char

Figura 2-10: Secuencia de Trama MODBUS RTU

Es preciso aclarar que entre dos tramas consecutivas no se suma el t3.5 de fin de una trama con el t3.5 de inicio de la siguiente. Esto queda reflejado en la siguiente figura:

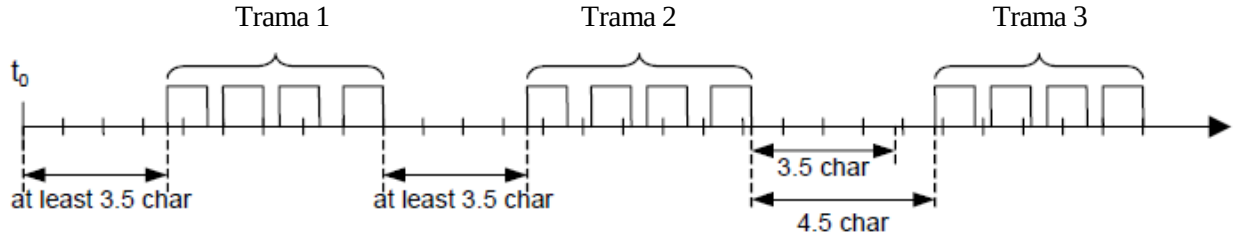

Figura 2-11: Separación de tramas MODBUS RTU

Tal como representa la siguiente figura dentro de cada trama los Bytes se tienen que transmitir de manera continua. Si entre dos Bytes se produce un silencio de más de 1.5 veces el tiempo de carácter se considera errónea la trama y se descarta.

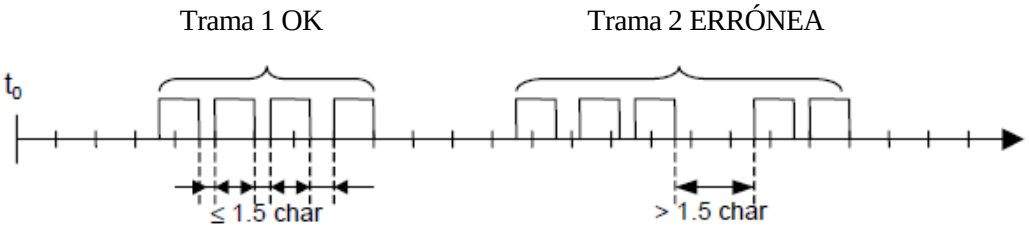

Figura 2-12: Separación de bytes en tramas MODBUS RTU

2.3.8 Modo de transmisión *ASCII*

Como se puede ver en la figura existen dos modos dependiendo de si se comprueba la paridad o no. El modo definido por defecto por el estándar MODBUS es el de paridad par, quedando los modos de paridad impar o no paridad como opcionales.

Los bits se transmiten en el siguiente orden: de izquierda a derecha; desde el bit menos significativo (LSB) hasta el más significativo (MSB).

En el modo de no paridad como se puede observar se sustituye dicho bit por otro bit de stop.

Con chequeo de paridad

Inicio	1	2	3	4	5	6	7	Par	Stop

Sin chequeo de paridad

Inicio	1	2	3	4	5	6	7	Stop	Stop

Figura 2-13: Secuencia de Bits en modo ASCII

La gran desventaja del modo de transmisión *ASCII* estriba en que un byte de información se codifica en dos caracteres *ASCII*, ocupando cada carácter *ASCII* 7 bits, más los bits extra que se necesitan de alineación y paridad de forma que un carácter *ASCII* ocuparía en total 10 bits. Si esto se compara con el modo *RTU* podría interpretarse erróneamente que el modo *ASCII* fuese más eficiente que el modo *RTU* que ocupa 11 bits por byte, nada más lejos de la realidad puesto que *ASCII* necesita enviar dos caracteres para transmitir un byte de

información, lo que hace en total 20 bits, sin embargo, en modo *RTU* el byte se envía con 11 bits.

El hecho de que se use el modo *ASCII* responde a cuestiones relacionadas con los medios físicos y la capacidad de los dispositivos usados, de manera que no sean compatibles con las especificaciones de los temporizadores definidos para MODBUS *RTU*.

A diferencia del modo *RTU*, en modo *ASCII* se permiten intervalos de hasta un segundo entre caracteres. A no ser que el usuario haya configurado un temporizador mayor, se considerará erróneo un intervalo de más de un segundo entre caracteres. En algunas aplicaciones de red de área extensa se usan intervalos de entre 4 o 5 segundos.

En modo *ASCII* las tramas se delimitan con caracteres en lugar de silencios: el carácter de inicio es "dos puntos" (:); en *ASCII* (0x3A) y los de final de trama el "retorno de carro" (CR) y "salto de línea" (LF); en *ASCII* (0x0D y 0x0A).

Los caracteres permitidos en todos los campos de la trama son hexadecimales 0-9, A-F codificados en *ASCII*. Los dispositivos monitorizan el bus continuamente hasta que encuentran el carácter ":". Entonces decodifican los caracteres siguientes hasta que detectan el fin de trama.

A continuación se muestra una trama MODBUS *ASCII*:

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

Figura 2-14: Trama MODBUS ASCII

La siguiente tabla muestra la misma trama enviada en cada modo y la diferencia en bits entre cada uno de ellos:

Campo	Hexadecimal	ASCII	Caracteres ASCII (10 bits)	RTU	Bytes RTU (11 bits)
Cabecera		:	1		
Dirección del esclavo	07	0 7	2	0000 0111	1
Código de función	03	0 3	2	0000 0011	1
Dirección de inicio (Alto)	00	0 0	2	0000 0000	1
Dirección de inicio (Bajo)	0A	0 A	2	0000 1010	1
Nº de registros (Alto)	00	0 0	2	0000 0000	1
Nº de registros (Bajo)	02	0 2	2	0000 0010	1

Chequeo de errores		LRC	2	CRC	2
Fin de trama		CR LF	2		
Total de Bits			170		88

Tabla 2-3: Comparativa entre modo ASCII y RTU

2.3.9 Comprobación de errores

MODBUS define dos modos para la comprobación de errores:

Chequeo de paridad: Dentro de la trama, se aplica a cada byte que se transmite de manera que el transmisor calcula para cada byte el bit de paridad y lo adjunta al mensaje. El receptor cuando lo recibe calculará la paridad del mensaje y comprobará que coincida con el bit de paridad que el transmisor adjuntó al mensaje. En caso contrario ignora el mensaje. El esclavo no responderá al maestro y el temporizador del maestro expirará.

Este tipo de verificación sólo es eficaz en caso de que el número de bits erróneos sea impar ya que si fuese par no cambiaría la paridad del mensaje.

Si el modo seleccionado es el de no paridad, no se comprueba, lógicamente, y se sustituye dicho bit por otro bit de stop.

Chequeo de CRC / LRC: Se aplica a la trama completa. Es un algoritmo que aplicado por el transmisor a la trama (sin incluir delimitadores) da como resultado un campo de dos bytes que se incluye en la trama antes de enviarla, de esta manera el receptor aplica el mismo algoritmo y comprueba si el resultado coincide con el campo de CRC / LRC que el transmisor incluyó en la trama. En caso contrario se ignora el mensaje. El esclavo no respondería al maestro y es el maestro el que tomará la decisión oportuna cuando expire su temporizador.

2.3.10 Códigos de función

Ya que el campo de código de función mide un byte y el bit de mayor peso se usa para que el esclavo indique si ha habido error; devolviendo el mismo código que se le solicitó con el bit de mayor peso a “1” y un byte en el campo de datos indicando el código de error que ha tenido lugar, quedan disponibles 127 códigos de función. Cada código de función se corresponde con una operación diferente.

Existen tres categorías de códigos de función:

- **Públicos:** Estos códigos de función están bien definidos, aprobados, testeados y documentados por la comunidad MODBUS.
- **Definidos por el usuario:** Son implementaciones específicas que un usuario crea para su aplicación y que no está soportada por el estándar.
- **Reservados:** Algunas compañías tienen funciones reservadas para sus productos que no son accesibles públicamente.

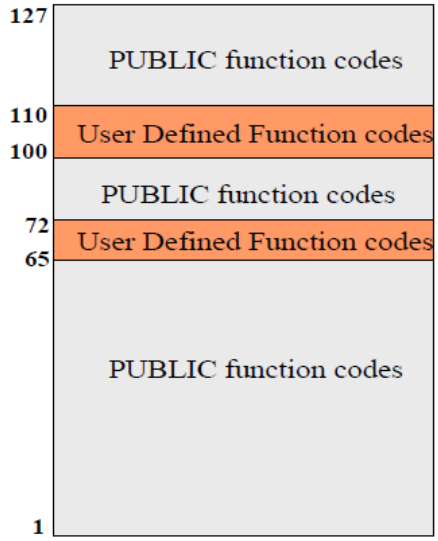

Figura 2-15: Categorías de códigos de función

Según el tipo de operación que se desea realizar sobre el esclavo se distinguen dos tipos:

Lectura / escritura: Sirven para consultar o escribir datos en la memoria del esclavo.

Control: Permiten realizar algunas acciones sobre el esclavo. Por ejemplo, ponerlo en modo de sólo escucha.

La siguiente tabla muestra las funciones más usadas:

Command	Function Code
01	Read Coils
02	Read Discrete Inputs
03	Read Holding Registers
04	Read Input Registers
05	Write Single Coil
06	Write Single Register
07	Read Exception Status
08	Diagnostics

Tabla 2-4: Funciones MODBUS más usadas

A continuación se van a describir las funciones a través de ejemplos que hagan más clara la explicación. Las funciones de lectura se pueden agrupar dos a dos ya que son completamente equivalentes, la única diferencia estriba en el tipo de dato leído; uno se puede también escribir y otro no, pero el tamaño es el mismo.

Read Coils (01) – Read Discrete Inputs (02)

Estas dos funciones sirven para que el maestro pueda consultar el estado de una cantidad, que se pasa como dato en la trama, determinada de coils o entradas contiguas en el esclavo. Pongamos como ejemplo la siguiente trama codificada en hexadecimal para ahorrar espacio, donde el maestro consulta el estado de los 10 primeros coils del esclavo con dirección 16:

Dirección esclavo	Código de función	Dirección de inicio	Cantidad de bits	CRC
10	01	00 00	00 0A	----

Como se puede apreciar la dirección del esclavo se corresponde con el número 16 en decimal, la dirección de inicio indica la dirección a partir de la cual se quiere consultar la información, está referenciada a cero, y la cantidad de bits que se van a leer se indica en el campo siguiente también en hexadecimal, correspondiendo al valor deseado 10 en decimal. El CRC no se ha calculado por simplificar, pudiéndose consultar el algoritmo para calcularlo en la especificación del estándar.

A continuación se muestra la respuesta del esclavo a la consulta anterior:

Dirección esclavo	Código de función	Bytes de respuesta	Estado de los bits	CRC
10	01	02	FF 03	----

En la respuesta se envía el código del esclavo para que el maestro sepa de quién es la respuesta y el mismo código de función. Si hubiese ocurrido algún error el código de función sería 0x81, esto es, 0x01 + 0x80 ya que 0x80=10000000b, que como se dijo, poner el MSB a "1" es la forma de indicar que ha habido un error.

En el campo bytes de respuesta se indica la cantidad de bytes que se han leído; si el número de bits solicitados no es múltiplo de 8 sobrarán bits que se rellenarán con ceros hacia el MSB. Esto se entiende mejor a través del ejemplo: en nuestro caso se pidieron 10 bits, vamos a suponer que el estado de esas diez entradas es "1", los dos bytes tomarían los valores hexadecimales 0xFF03, que en binario serían:

11111111 00000**11**

Se han marcado en negrita los bits que se corresponden con el estado de las entradas, quedando los ceros de relleno para completar el segundo byte.

Real Holding Registers (03) – Read Input Registers (04)

Estas dos funciones son totalmente análogas a las anteriores pero en lugar de leer entradas de un bit, leen registros de dos bytes. Un ejemplo:

Dirección esclavo	Código de función	Dirección de inicio	Nº de registros	CRC
10	03	00 00	00 0A	----

En este caso la dirección de inicio indicada es la misma que en el caso anterior, y al ser diferente el tipo de dato que queremos leer podría parecer erróneo el hecho de que hagamos referencia a la misma dirección de memoria para intentar leer datos diferentes. Esto no es así porque el código de función lleva implícita la información que el dispositivo necesita para conocer a qué tipo de dato nos estamos refiriendo y como se puede ver en la Figura 3; para cada tipo de dato hay un rango de direcciones aunque todas empiezan en cero.

La respuesta del esclavo es evidente por analogía con la función explicada anteriormente.

Write Single Coil (05)

Esta función se usa para modificar el estado de una salida tipo coil. Se pasa la dirección que se quiere modificar, seguida del estado: 0x0000 para "0" y 0xFF00h para "1". Por ejemplo:

Dirección esclavo	Código de función	Dirección del coil	Valor a escribir	CRC
10	05	00 00	FF 00	----

Como ya se ha explicado en el apartado anterior se usa la misma dirección para el coil (el primero), no queriendo decir esto que ocupe la misma dirección de memoria que los datos leídos con las funciones anteriormente descritas, ya que estamos haciendo referencia al bloque de memoria de los coils.

La respuesta del esclavo sería un eco de la solicitud, esto es, idéntica si se ha escrito sin errores para confirmar esclavo, dirección y valor escrito. En caso de error se suma 0x80 al código de función, como ya se explicó anteriormente, resultando el código de la respuesta 0x85.

Write Single Register (06)

Esta función es análoga a la anterior pero en lugar de escribir un bit escribe un registro, es decir, 2 bytes. Un posible ejemplo:

Dirección esclavo	Código de función	Dirección del registro	Valor a escribir	CRC
10	06	00 00	AA 05	----

La respuesta del esclavo en caso de que todo haya ido bien sería un eco de la solicitud igual que en el caso anterior.

2.4 Arduino

Arduino surge en 2005 como un proyecto de hardware libre para estudiantes del instituto Ivrea, en Ivrea (Italia). Uno de los fundadores, el profesor Massimo Banzi, reconoce que en ningún momento Arduino surgió como idea de negocio sino que fue una forma de evitar que el proyecto se perdiese ante el inminente cierre del instituto. De forma que si creaban un proyecto de hardware abierto, no podría ser embargado.

La idea del proyecto era acercar la electrónica y la programación de sistemas embebidos a todo el mundo, gracias a sus placas de bajo coste, siendo el primer objetivo para la producción en serie que su precio no superase los 30 euros.

En 2005 se incorporó al proyecto Tom Igoe, un reputado profesor de computación física que ayudó notablemente a desarrollar el proyecto a gran escala y distribuir, gracias a sus contactos, las placas por territorio estadounidense.

Las placas Arduino están disponibles ensambladas o en forma de kits para su montaje. Además, el hecho de que los esquemáticos sean públicos hace que cualquiera con conocimientos básicos en la materia pueda montarse su propia placa. En 2013 se estimó que se habían vendido 700.000 placas oficiales, por lo que la comunidad Arduino era ya un hecho, factor fundamental en su éxito, pues es muy fácil encontrar información de calidad tanto para usuarios principiantes como para usuarios avanzados.

Actualmente existen multitud de modelos de Arduino oficiales y no oficiales; desde las propias placas de desarrollo hasta placas de expansión, accesorios, kits e incluso impresoras 3d. Las placas de desarrollo Arduino han ido evolucionando con el tiempo. La original usaba un microcontrolador ATmega AVR de 8 bits, pero actualmente las hay con controladores ARM de 32 bits para tareas más pesadas, ambos desarrollados por la compañía Atmel. La más básica es la Arduino UNO y se puede encontrar a un precio que ronda los 20 euros.

3 COMPONENTES DEL SISTEMA

La mayoría de los componentes usados en este proyecto han sido adquiridos en la tienda online de Libelium, empresa aragonesa con sede en Zaragoza, que sigue la filosofía de Arduino aunque con propósitos lógicamente comerciales; acercar la tecnología al gran público con numerosos tutoriales y con un foro en inglés y español en su página web dónde los usuarios comparten conocimiento.

3.1 Arduino UNO Rev3

Es la versión más actual de la placa Arduino original y la más vendida ya que sus prestaciones son suficientes para afrontar la mayoría de proyectos que existen en la red y su precio no es alto.

Sus principales características son:

- Microcontrolador Atmega328
- Voltaje de entrada 7-12V
- 14 pines digitales de Entrada/Salida (6 de ellos PWM)
- Corriente máxima por pin 40 mA tanto de entrada como de salida
- 6 entradas analógicas conectadas a un conversor A/D de 10 bits
- 32KB de memoria flash (0.5KB ocupados por el bootloader)
- SRAM 2KB
- EEPROM 1KB
- Reloj de 16 MHz

Algunos de los pines tienen características especiales:

- **RXD y TXD:** Pines 0 (receptor) y 1(transmisor), se usan para transmisiones serie en modo TTL.
- **Pines de interrupción externa:** Son los pines 2 y 3, y pueden configurarse para generar interrupciones en el Atmega activadas de dos formas:
    - A nivel bajo, esto es, cuando se da como entrada un cero lógico.
    - Por flanco, esto es, cuando el pin cambia de nivel alto a bajo o viceversa.
- **PWM:** Tiene disponible 6 pines PWM que permiten generar salidas analógicas de hasta 8 bits.
- **SPI:** Los pines 10, 11, 12 y 13 sirven para realizar comunicaciones SPI que permiten transmitir información en modo full dúplex entre maestro/esclavo.
- **I²C:** Permiten realizar comunicaciones en bus I²C soportada por la mayoría de dispositivos vendidos tales como LCD'S, sensores, memorias, etc.

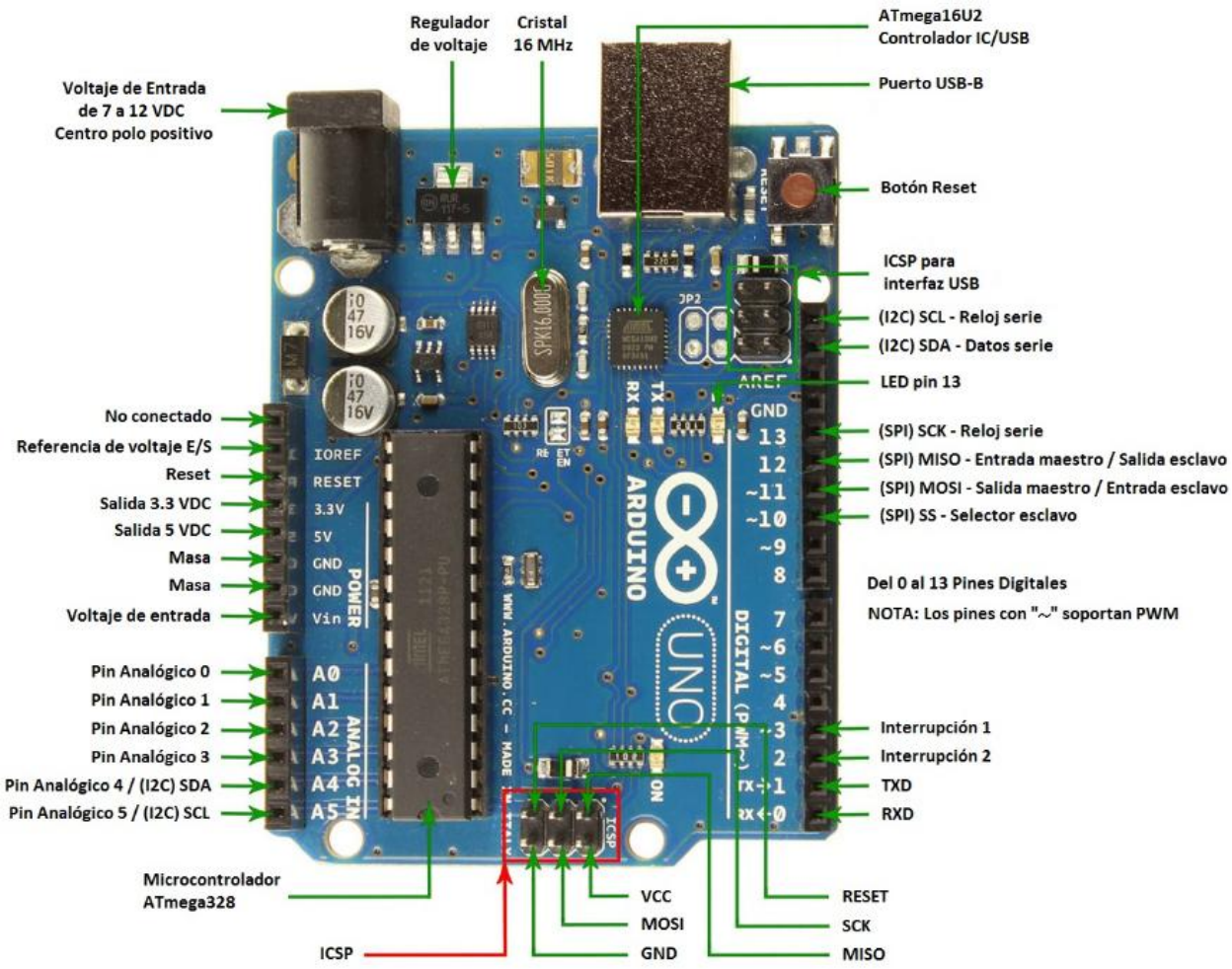

Figura 3-1: Placa Arduino UNO Rev3

3.2 Módulos / Shields

Los módulos o shields son placas de expansión que se montan unas encima de otras y sirven para ampliar el hardware y, por ende, las capacidades y funcionalidades de nuestro Arduino. Son apilables, por lo que se pueden montar varios módulos sobre una misma placa Arduino.

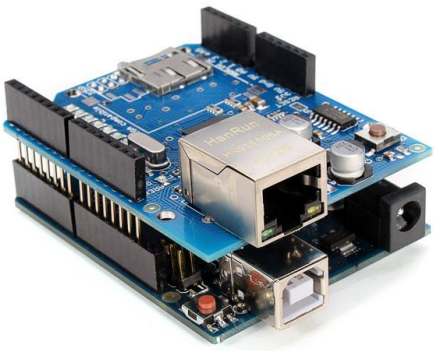

Figura 3-2: Arduino con Módulo Ethernet acoplado

3.2.1 Módulo Multiprotocol

El módulo Multiprotocol Radio Shield es un módulo diseñado para conectar módulos de comunicación a la placa Arduino. Incluye las conexiones para el bus SPI que permiten conectar hasta dos módulos de comunicaciones al mismo tiempo. Pudiendo usar dos formas de comunicación diferentes en nuestro proyecto usando solamente una placa Arduino, por ejemplo RS-485 y RFID tal como muestra la figura 18.

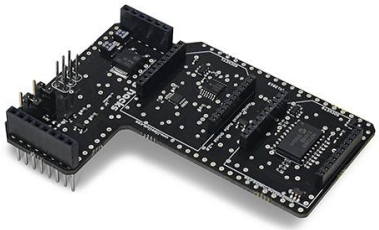

Figura 3-3: Módulo Multiprotocol Radio Shield

Figura 3-4: Arduino con Módulos RS-485 y RFID

3.2.2 Módulo MODBUS/RS-485

Éste módulo permite usar el estándar RS-485 y MODBUS para comunicarse con otros dispositivos que lo soporten. Se puede conectar a otros dispositivos de dos formas: usando un par de cobre trenzado conectado a la bornera o usando un cable DB9.

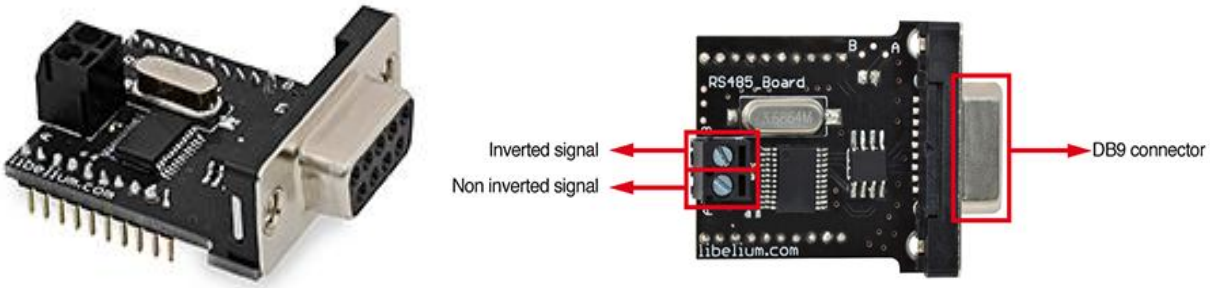

Figura 3-5: Módulo MODBUS/RS-485

3.3 Sensores y actuadores

Los sensores son los dispositivos con los que se tomarán las medidas de los parámetros físicos que serán

consultados por el maestro. Generalmente estos sensores estarán repartidos por el lugar que se desea monitorizar, por ejemplo, una habitación con varios sensores de presencia y alguno de temperatura para controlar en todo momento la estancia.

Sensor PIR: el sensor PIR toma nombre de sus siglas en inglés de sensor infrarrojo pasivo. Está basado en un sensor piroeléctrico que detecta la radiación infrarroja que emiten todos los objetos en su campo de visión y su comparación en el tiempo; si cambia, lo notifica a través de un pin digital de salida. Se llama pasivo porque no emite radiación alguna.

Están cubiertos de una cápsula que no es más que una lente de Fresnel para aumentar su campo de visión.

Figura 3-6: Sensor PIR

La siguiente figura muestra su principio de funcionamiento:

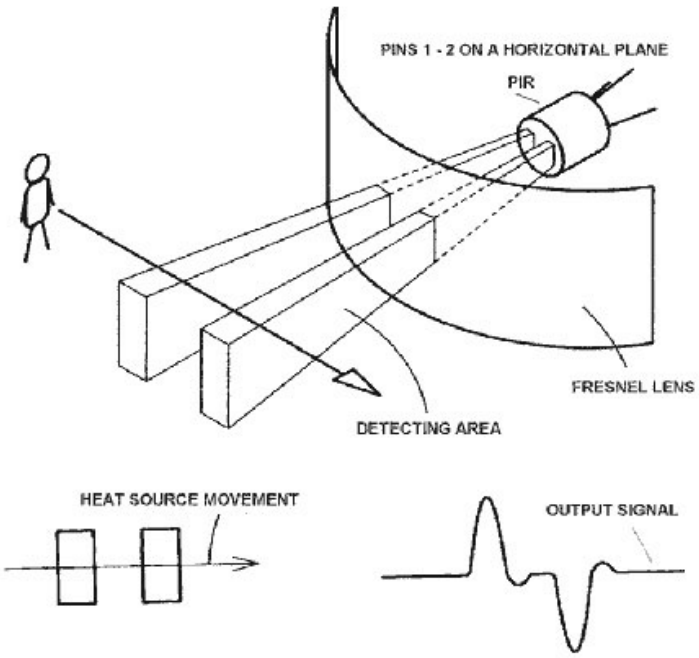

Figura 3-7: Funcionamiento de un sensor PIR

Como se ve en la figura cuando una fuente de calor se mueve, el sensor detecta la variación de la radiación infrarroja recibida y emite una señal digital en su pin de salida.

Servo: es un dispositivo actuador que es capaz de girar un cierto ángulo que depende de la señal de control y mantenerse en dicha posición.

Figura 3-8: Servo

3.4 Otros componentes electrónicos

Además se ha usado para hacer las conexiones una placa de pruebas con sus correspondiente set de cables, un led y un potenciómetro.

Placa de pruebas o protoboard: del inglés prototype board, se usa para conectar componentes de forma sencilla sin necesidad de soldarlos. Como su nombre indica sirve para hacer pruebas antes de fabricar algún componente electrónico de forma definitiva. Las ha de muchos tipos pero todas siguen la misma lógica; tiene una serie de filas que están unidas eléctricamente de manera que podemos conectar los componentes pinchándolos en la placa y crear el circuito que queramos para probarlo de forma provisional.

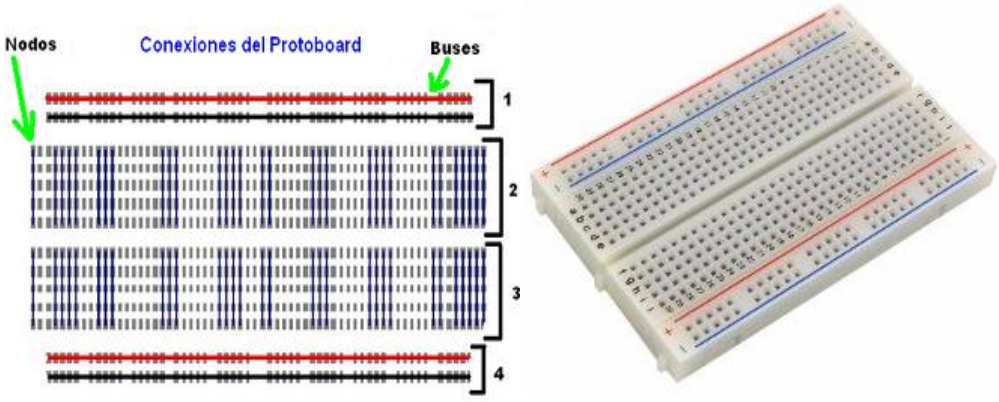

Figura 3-9: Conexiones de una placa de pruebas

Figura 3-10: Cables para placa de pruebas

Como puede verse en la figura sólo los puntos unidos por los segmentos se encuentran unidos eléctricamente. De esta manera, aunque los segmentos de las columnas centrales (2,3) sean del mismo color, se encuentran eléctricamente aislados.

Led: Un led (Light-emitting diode) no es más que un diodo que polarizado en directa, emite energía en forma de fotones.

Figura 3-11: Led

Potenciómetro: Un potenciómetro es una resistencia variable que se puede ajustar manualmente. Tiene tres terminales, los extremos (A,C) se conectan a la diferencia de potencial que se desea regular y en el terminal central (B), también llamado cursor, tenemos la salida que podemos controlar.

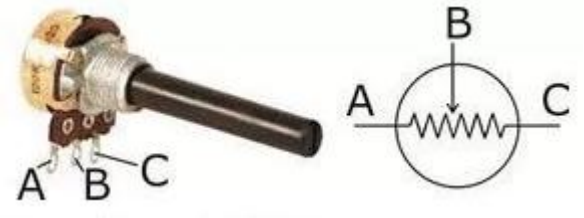

Figura 3-12: Potenciómetro

4 DESARROLLO Y PUESTA EN MARCHA

4.1 Instalación del IDE de Arduino

Para descargar el entorno de desarrollo de Arduino tan sólo hay que entrar a la sección de descarga del software en la página oficial del proyecto:

http://arduino.cc/en/main/software

La primera versión que aparece para descargar es la más reciente que a la fecha de escribir este documento es la 1.8.3.

Elegimos la opción adecuada según el sistema operativo, en este caso Windows. Aparecen dos opciones: "Installer" y "Zip files". Conviene elegir la primera ya que de esta forma se instalarán los drivers automáticamente.

Una vez finalizada la descarga hacemos doble clic en el instalador con extensión .exe y comenzará el proceso de instalación. El asistente nos preguntará por los componentes que deseamos instalar, seleccionamos todos como muestra la siguiente figura:

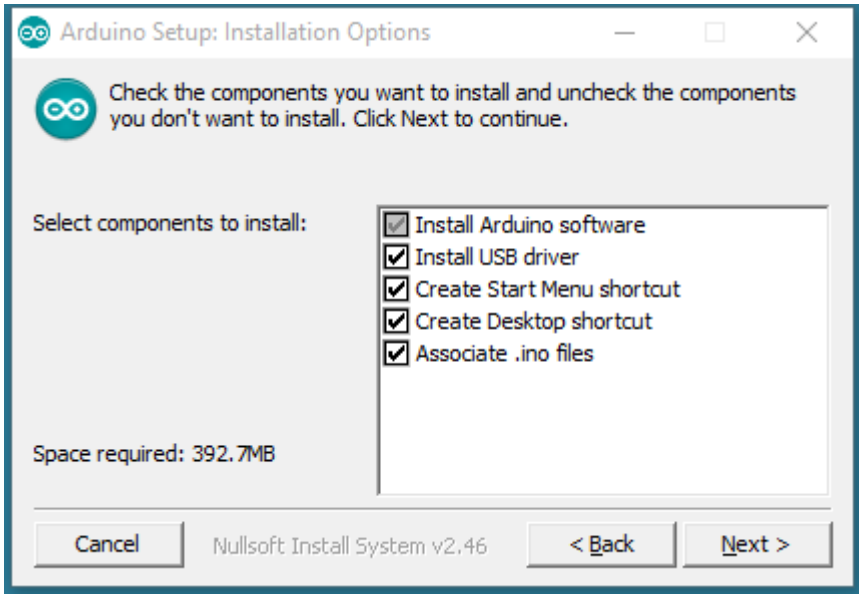

Figura 4-1: Opciones de instalación del IDE Arduino

A continuación se elige el directorio de instalación, lo normal es dejar el que viene por defecto:

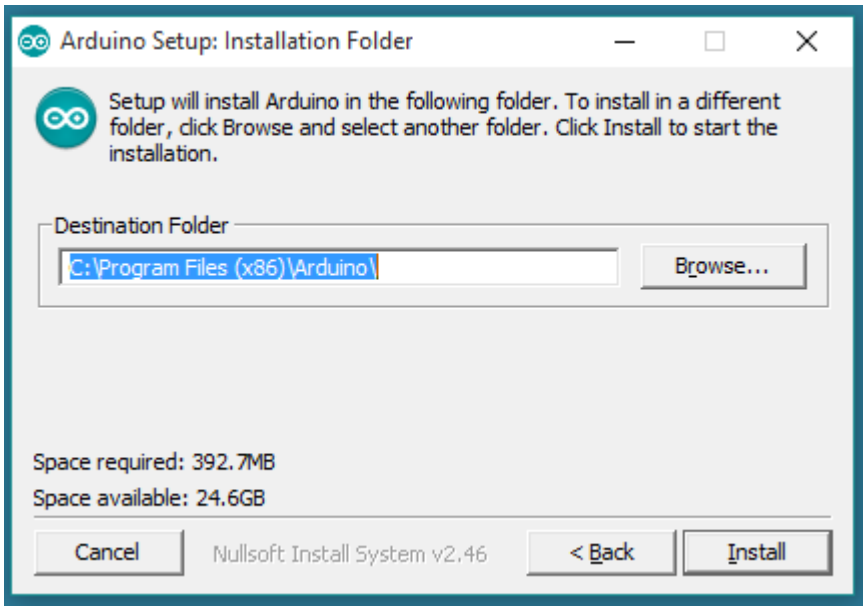

Figura 4-2: Carpeta de instalación

Por último esperamos a que finalice el proceso de instalación:

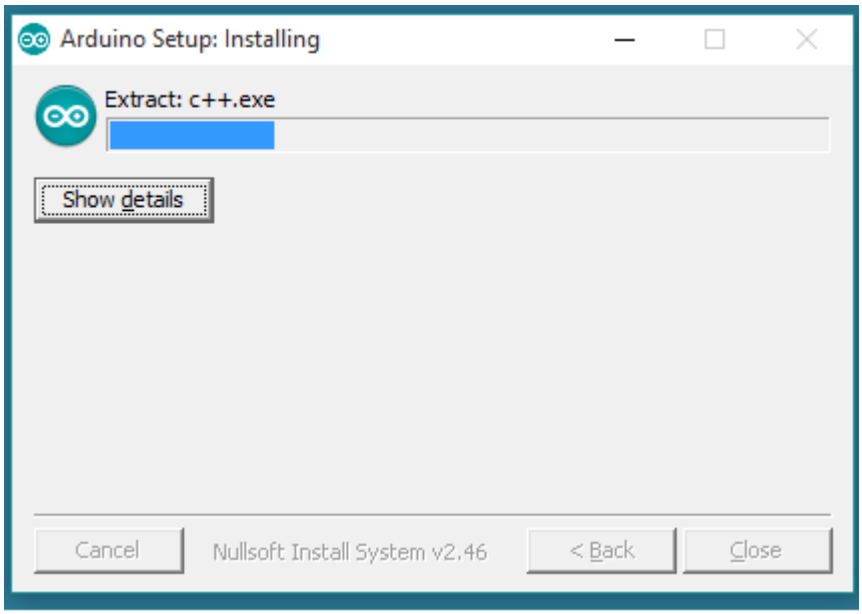

Figura 4-3: Instalación en proceso

Una vez finalizada la instalación del IDE desde dónde se programará la placa ya podemos conectarla a nuestro PC a través de un cable USB tipo A-B, conocidos generalmente por ser los cables que se usan para conectar las impresoras. Este cable además de servir para comunicar el ordenador con el Arduino también sirve para alimentarlo, por lo que no será necesario conectar una fuente de tensión al jack específico. Una vez conectado se debe de encender el led de Power en la placa, lo cual quiere decir que tiene alimentación.

La primera vez que se conecta el Arduino al ordenador, Windows lo detectará e instalará sus drivers automáticamente, a continuación ya estamos listos para hacer doble clic y entrar en el IDE de Arduino.

Se abrirá una ventana dónde automáticamente aparece un sketch para que empecemos a escribir código.

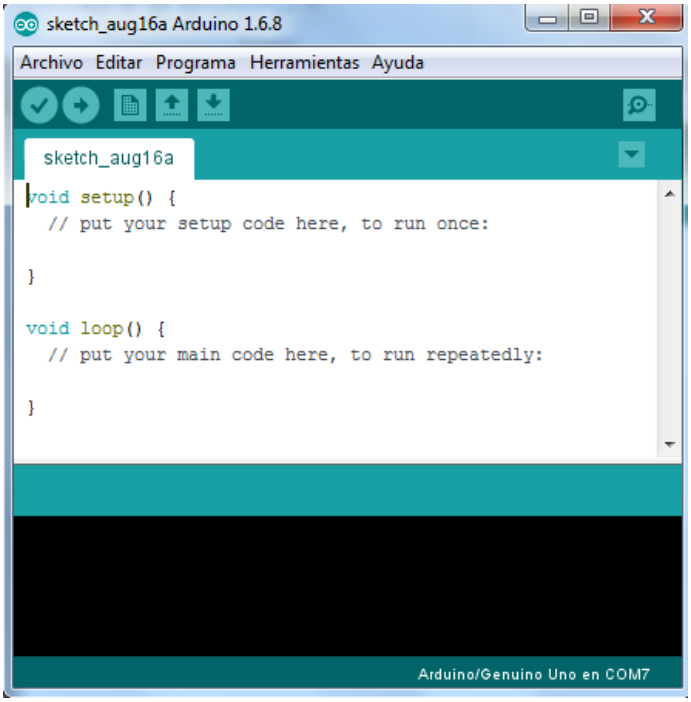

Figura 4-4: Ventana principal del IDE de Arduino

A continuación vamos a configurar el IDE para nuestra placa Arduino uno, hacemos clic en Herramientas y en Placa elegimos nuestro modelo en la lista desplegable:

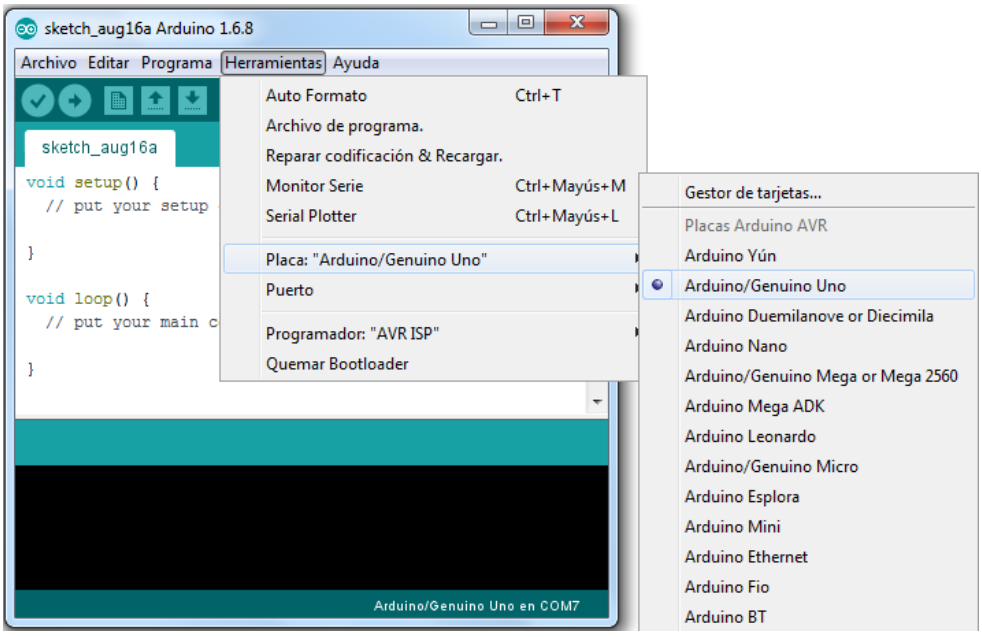

Figura 4-5: Seleccionando modelo de placa

Ya sólo falta comprobar que tenemos un puerto asignado correctamente a nuestra placa. Debe aparecer así:

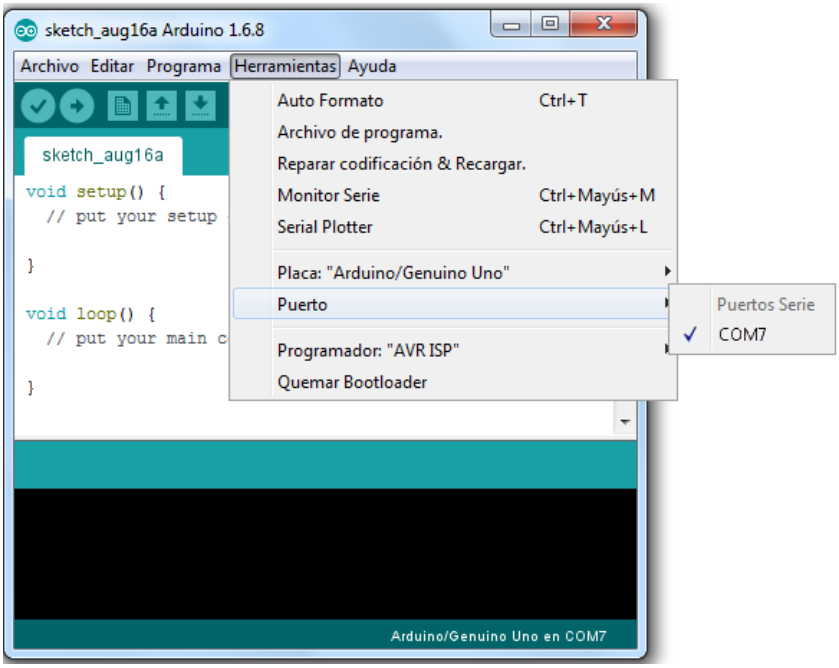

Figura 4-6: Puerto COM7 asignado

El número de puerto puede cambiar de un ordenador a otro.

Ya estamos en disposición de cargar un ejemplo en la placa para comprobar que todo funcione correctamente. Se va a cargar uno de los ejemplos que incluye el IDE; se llama blink que en inglés es parpadeo y lo único que hace es hacer parpadear un led. Lo elegimos de esta forma:

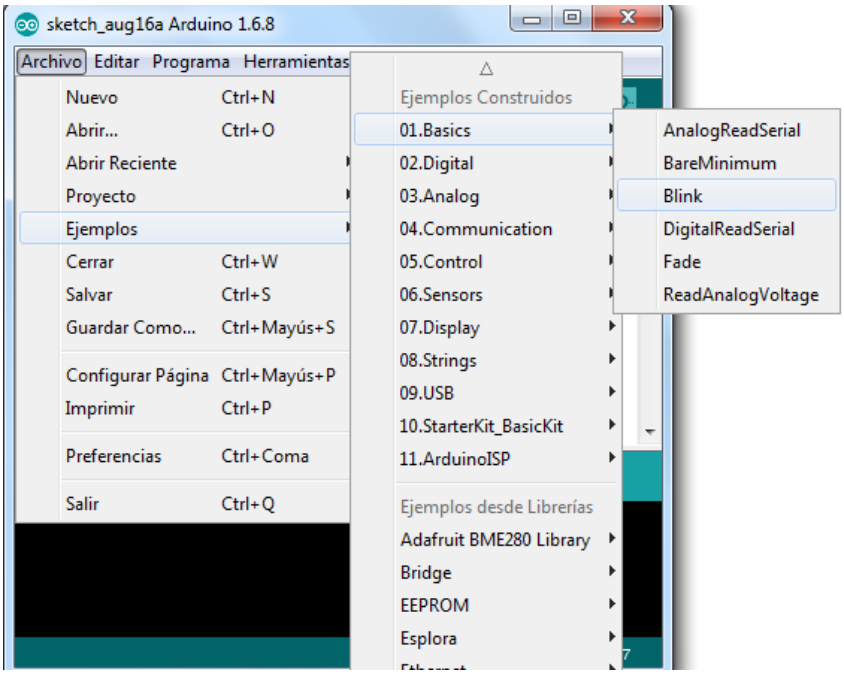

Figura 4-7: Seleccionando el ejemplo Blink

Una vez seleccionado ahora hay que cargarlo en la placa, para ello hay que pulsar la flecha que se ve en la siguiente figura, si ponemos el ratón encima se puede leer su función:

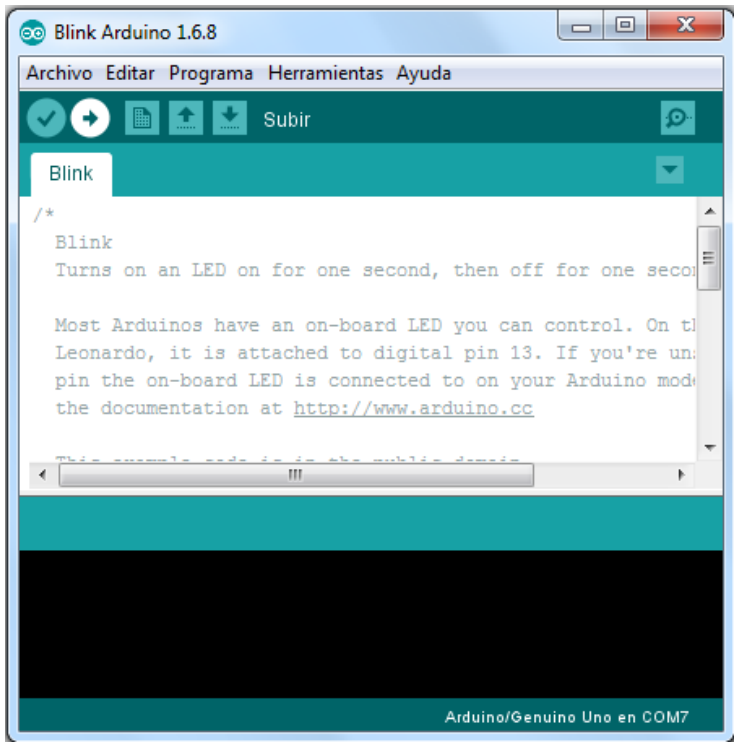

Figura 4-8: Botón para cargar un programa en la placa

Si todo ha ido bien debe aparecer el siguiente mensaje en la parte de abajo de la ventana:

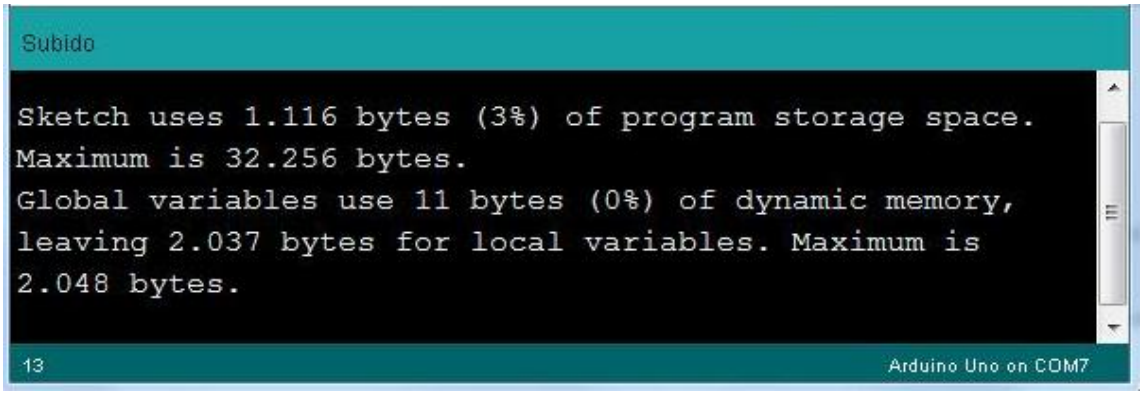

Figura 4-9: Programa cargado sin errores

Y Arduino empezará a ejecutar el programa, por lo que el led debería empezar a encenderse y apagarse. Una vez conseguido esto ya estamos seguros de que nuestro IDE y nuestra placa funcionan correctamente.

4.2 Instalación de librerías de los sensores

Para instalar cualquier librería siempre se procede de la misma forma:

- Descargar la librería correspondiente de la web del fabricante.
- Descomprimirla; ya que estará comprimida la mayoría de las veces.
- Pegar la carpeta de la librería en el directorio donde se guardan las librerías del IDE de Arduino; que si se instaló en la carpeta por defecto será C:\Program Files\Arduino\libraries.

Las librerías de los sensores que se han usado son:

ArduinoRS485-MODBUS-library-v0_4: librería creada por Libelium que permite gestionar la comunicación vía MODBUS usando programación C++ de alto nivel, de forma que la creación de la trama a bajo nivel resulta casi transparente al usuario.

Al instalar esta librería se encontraron algunos errores que se solucionaron cambiando en el archivo rs485.cpp el include /...spi/spi.h por <spi.h>

Servo: librería incluida por defecto en el IDE, usada para manejar el servo.

ArduinoUtils: librería propiedad de Libelium, es referenciada en la librería de MODBUS comentada anteriormente. Da soporte a funciones para gestionar algunos dispositivos como el módulo multiprotocol.

4.3 Conexión de componentes

Se van a distinguir dos escenarios, ya que para conseguir el objetivo final se fue haciendo un acercamiento progresivo: en primer lugar sólo se intentó comunicar un maestro y un esclavo porque de esta forma era más fácil depurar e identificar posibles errores.

Es importante aclarar que la conexión entre maestro y esclavo se hace a través de la bornera del módulo mostrado en la Figura 19, puesto que se intentaron conectar a través de un cable DB9 y no fue posible porque el cable que proporcionaba el fabricante estaba diseñado para conectar un dispositivo transmisor con otro receptor por lo que no estaba cruzado, lo cual quiere decir que el pin transmisor de un dispositivo iba a parar a su homólogo en el otro dispositivo, cuando para conectar dos dispositivos transmisores es necesario que el pin transmisor se conecte al receptor del otro dispositivo y viceversa. Una posible solución sería abrir el cable y cruzar los hilos manualmente pero teniendo la opción de la bornera se tomó ésta por ser más sencilla.

Primer escenario: maestro-esclavo

En este caso el maestro está conectado únicamente al esclavo por el par trenzado que une los dos terminales de cada bornera etiquetados como A y B con sus homólogos en el otro dispositivo, y a la placa de pruebas a través del cable rojo que se ve en la Figura que une el pin Vcc de su cabecera ICSP con la placa (para más detalles de los pines de Arduino, consultar Figura), de esta forma conseguimos tener una línea con tensión Vcc en la placa de pruebas para alimentar los demás componentes y otro cable negro que sale del pin Gnd que nos va a proporcionar la línea de tierra en la placa.

En la siguiente figura se muestra el cableado:

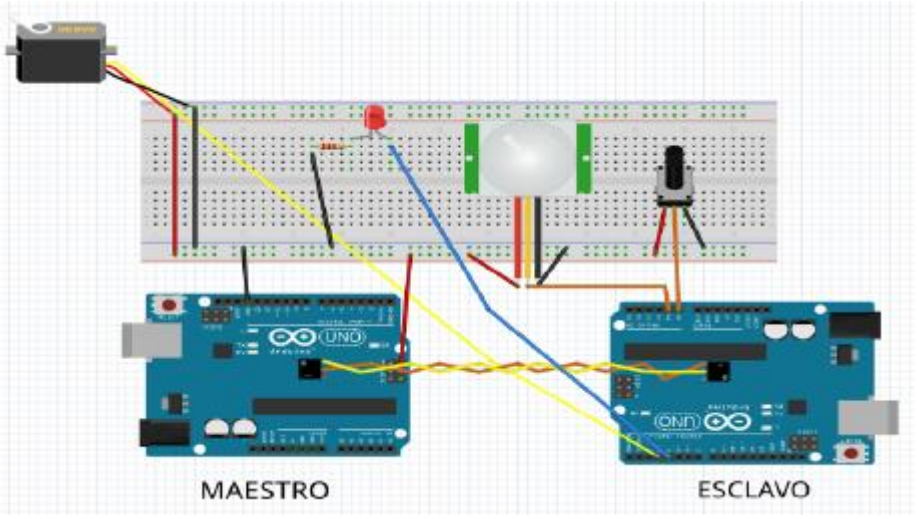

Figura 4-10: Cableado del primer escenario

Todos los sensores y actuadores van conectados a un pin de alimentación y a otro de tierra, aunque el led se encuentra en serie con una resistencia para limitar su corriente. La descripción detallada de las patas de los componentes se puede consultar en el capítulo Componentes del Sistema, no obstante, si se observa la línea de alimentación de la placa de pruebas comentada anteriormente y la que está a su lado que es la de tierra se puede ver fácilmente qué pines son los que llevan la alimentación y la tierra en los sensores. Además, los pines de los componentes conectados a tierra están conectados con cables de color negro. Los pines restantes (en los componentes de tres patas), que no están conectados ni a Vcc ni a tierra, proporcionan la medida tomada y están conectados al esclavo en pines destinados a entradas analógicas: el A1 para el sensor PIR y el A0 para el potenciómetro. Por último, los actuadores están conectados a salidas digitales. El caso del servo es algo especial, puesto que está conectado a una salida pwm necesaria para su manejo.

Segundo escenario: maestro-esclavos

La diferencia fundamental en este escenario con respecto al anterior es que se añade un segundo esclavo al montaje. Para conectarlos en bus se ha conectado el nuevo esclavo a la bornera del maestro, donde ya estaba conectado el primer esclavo, de manera que a la bornera del maestro llegan 4 cables. De esta forma, como el maestro queda en el centro se reduce la distancia máxima que recorre un mensaje hasta llegar a su destino a la mitad respecto a la que se tendría si se pusiese en un extremo y menores distancias implican menores pérdidas.

La conexión de los sensores y actuadores con los Arduino se realiza de la misma forma. Lo importante es elegir pines analógicos para los sensores y salidas digitales para los actuadores, con la particularidad de que además para el servo la salida deberá ser PWM.

En la siguiente figura se muestra el cableado del montaje:

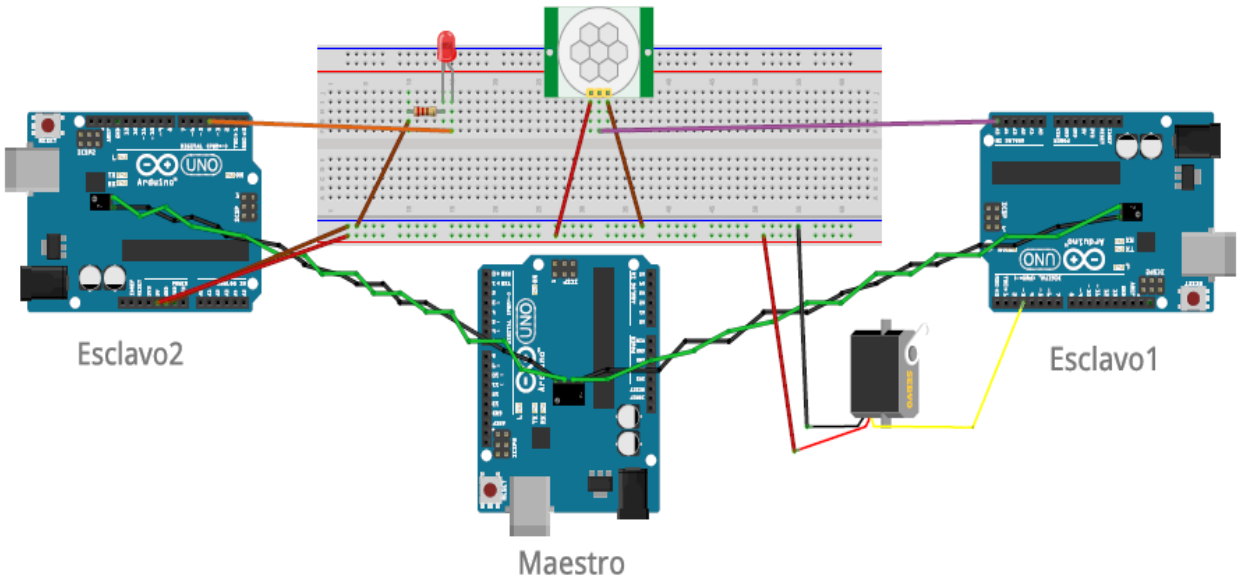

Figura 4-11: Cableado del segundo escenario

4.4 Funcionamiento

4.4.1 Descripción funcional

Primer escenario: como ya se ha comentado, el esclavo tiene conectados un sensor PIR y un potenciómetro. Arduino guarda la salida de esos dos componentes en registros MODBUS que van a ser consultados de forma periódica por el maestro. Las variables asociadas al potenciómetro y al sensor PIR tomarán valores entre 0 y 1023 ya que el convertidor A/D es de 10 bits. Éstos valores se mapearán a rangos entre 0 y 255 ya que el PWM de Arduino es de 8 bits; el servo podrá tomar cualquier valor en ese rango de forma que dicho valor

determinará el ángulo de giro. El led sólo podrá tomar los valores 0 para apagado y 255 para iluminado ya que no se conectará a una salida PWM.

En definitiva, moviendo el potenciómetro controlaremos el ángulo de giro del servo y el led se encenderá cuando el sensor PIR detecte movimiento.

Segundo escenario: en este caso el "esclavo1" tiene el sensor PIR y el servo conectados mientras que el "esclavo2" se conecta a un led. La diferencia con respecto al caso anterior es que la detección de movimiento del sensor PIR hará que el maestro accione los actuadores de cada uno de los dos esclavos. Partiendo del estado de reposo el maestro realiza consultas constantemente sobre el estado del sensor PIR. Cuando la respuesta del esclavo es que el sensor ha detectado movimiento el maestro envía la orden de encender un led al "esclavo2" y la orden de girar noventa grados el servo al "esclavo1". Cuando en las próximas consultas el maestro detecte que deja de haber movimiento, entonces envía la orden de apagar el led y de girar el servo noventa grados en el sentido contrario. Como se aprecia en la figura de abajo, se repiten las ordenes cuando el estímulo del sensor se repite pero el servo no gira nunca más de noventa grados en el mismo sentido ya que la orden no es realmente de giro si no de situarse en un ángulo de +90 o -90 grados por lo que una vez que alcanza dicha posición repetir la petición no provocará movimiento alguno.

De esta forma podría ser este montaje apropiado para simular un paso a nivel con barrera o un control de acceso a un parking añadiendo algunos sensores más para que la barrera se abra dependiendo de si hay trenes pasando o sitios libres en el parking. Cabe destacar también que en estos casos el sensor PIR no sería la solución adecuada, ya que la barrera se bajaría si el coche se quedase parado un momento delante del sensor. Habría por tanto que sustituirlo por un sensor fotoeléctrico pero el código no cambiaría.

A continuación se van a mostrar las salidas por el monitor serie de los tres dispositivos en funcionamiento:

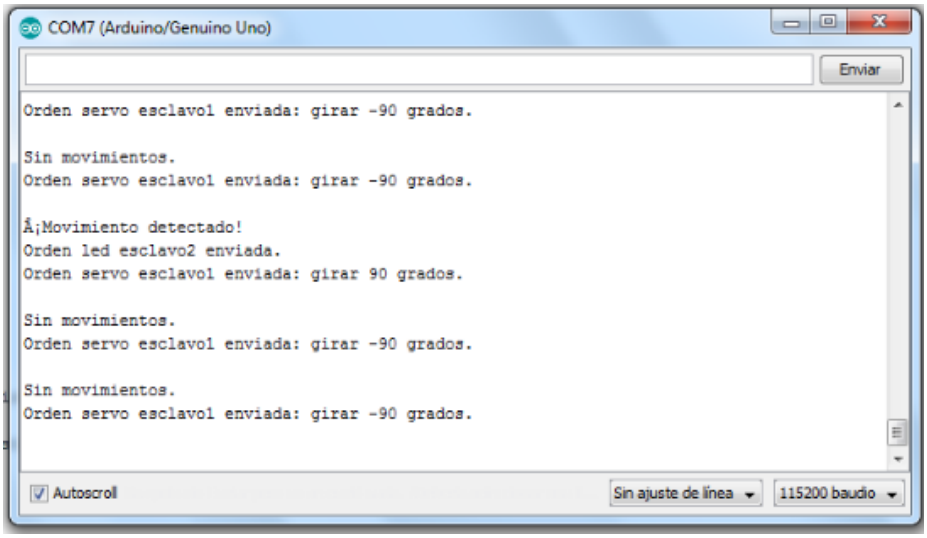

Figura 4-12: Salida monitor serie del maestro

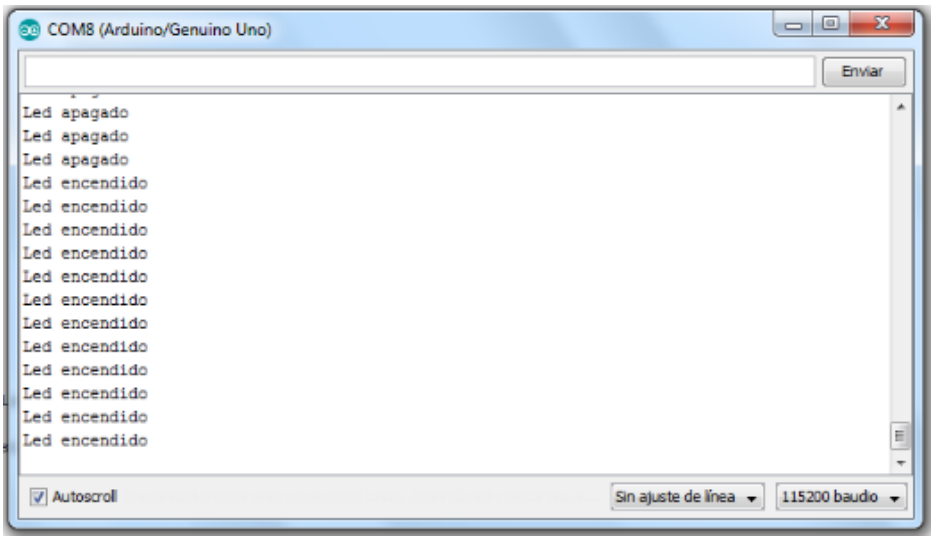

Figura 4-13: Salida monitor serie del esclavo2

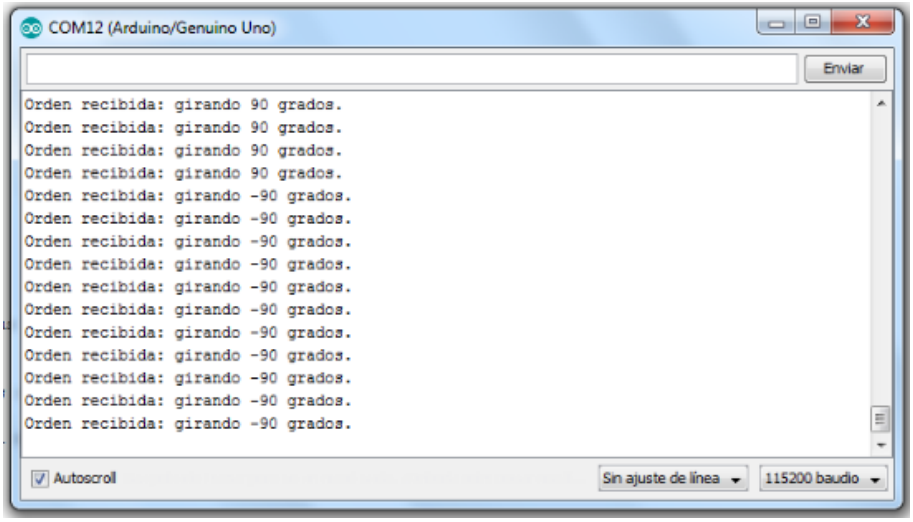

Figura 4-14: Salida monitor serie del esclavo1

4.4.2 Funciones usadas en el código

En este apartado se van a describir las funciones más importantes que se han usado en el código:

- **Setup( )**

  Es la primera función que se llama y se ejecuta una sola vez después del encendido o reseteo del Arduino. Se usa para iniciar variables, configurar los pines o usar algunas librerías antes de ejecutar el resto del programa.

- **Loop( )**

  Es como la función main en C, pero se ejecuta una y otra vez de forma consecutiva. Dentro de ella se escribe el código que se va a ejecutar en bucle para que Arduino desempeñe la tarea deseada.

- **Serial.begin(tasa)**

  Sirve para establecer una comunicación a través del puerto serie, en este caso vía USB, para comunicarse con el ordenador. Se pasa como argumento la tasa de baudios a la que se van a transmitir los datos. El valor típico es 9600 aunque también se permiten otras velocidades. El valor que se pase como argumento será el mismo que se deberá seleccionar para el monitor serie del IDE, de esta forma podremos ver por pantalla la información que se transmite.

- **delay(milliseconds)**

  Pausa el programa durante el tiempo que se le pase como argumento en milisegundos.

- **node.begin(tasa)**

  Node es un objeto de la clase MODBUSmaster485 y begin es un método que sirve para establecer la comunicación MODBUS RTU con uno o varios esclavos. Toma como argumento la tasa en baudios a la que se establecerá la comunicación.

- **Serial.print(cadena)**

  Sirve para imprimir en el monitor serie la cadena que se pase como parámetro.

- **Serial.println(cadena)**

  Igual que la anterior pero incluye un salto de línea al final de la cadena.

- **Node.readHoldingRegisters(dirección, nº de bloques)**

  Lee el número de registros pasado como segundo argumento a partir de la dirección indicada en el primero. Conviene recordar que los registros ocupan dos bytes y se direccionan desde cero.

- **map(valor, rango1inf, rango1sup, rango2inf, rango2sup)**

  Mapea el valor que se pasa como argumento de un rango inicial (rango1inf-rango1sup) a otro rango final (rango2inf-rango2sup). O dicho con otras palabras, es un cambio de escala.

- **node.writeSingleRegister(dirección, datos)**

  Escribe en el registro seleccionado a través de la dirección que se pasa como primer argumento los datos que se pasan en el segundo.

- **node.getReponseBuffer(índice)**

  Toma los datos que se han recibido como respuesta y se han guardado en el buffer de entrada. Se le pasa como argumento un índice que sirve para hacer referencia al bloque de datos que queremos leer dentro del buffer.

- **node.clearReponseBuffer( )**

  Limpia el buffer de respuesta. Es recomendable usarlo para asegurarse de que los datos que se leen cada vez son los correspondientes a la última respuesta que ha llegado.

- **Servo.atach(pin)**

  Configura el pin indicado para que se conecte el servo.

- **Servo.write(ángulo)**

  Hace que el servo gire el ángulo que se le pasa como argumento.

4.4.3 Análisis de las tramas intercambiadas

Primer escenario: maestro-esclavo

En este caso, por cada iteración del programa, es decir, cada vez que se ejecuta la función *loop* se realizan dos lecturas (potenciómetro y PIR) y dos escrituras (led y servo) por parte del maestro. Como cada solicitud recibe una respuesta del esclavo en total se van a intercambiar ocho mensajes. Vamos a analizar las cuatro correspondientes al sensor PIR; la solicitud y respuesta de lectura y sus equivalentes para la escritura puesto que las cuatro restantes serían totalmente análogas sólo variando las direcciones que se leen y se escriben.

Antes de detallar las tramas intercambiadas se va a describir el formato en que se transmiten los caracteres ASCII que componen los campos dentro de cada trama. Consultando el archivo MODBUSMaster485.cpp de

la librería ArduinoRS485, según el método `begin(void)` de la clase MODBUSMaster485 la comprobación de paridad está desactivada por lo que se finaliza la transmisión del carácter con dos bit de parada. De esta forma cada carácter ASCII se transmitiría de la forma siguiente:

Inicio	1	2	3	4	5	6	7	Stop	Stop

A continuación se muestra la trama correspondiente a la solicitud de lectura del sensor PIR:

Dirección esclavo	Código de función	Dirección de inicio	Nº de registros	CRC
01	03	00 00	00 01	----

La dirección del esclavo es la primera ya que sólo hay uno, aunque se podría haber elegido otra, pero es la que se ha definido en el código. El código de función es el correspondiente a la función *readHoldingRegisters* y el número de registros que se va a leer es uno en la dirección cero. El CRC no se ha calculado por simplificar ya que no es más que un algoritmo definido en el estándar que realiza una serie de operaciones a nivel de bits.

La respuesta del esclavo, en caso de no haber errores, tendría la siguiente forma:

Dirección esclavo	Código de función	Bytes de respuesta	Estado de los bits	CRC
01	03	02	00 00	----

En este caso se ha considerado que el sensor PIR no estaba detectando ningún movimiento, por lo que el valor guardado en el registro serían dieciséis bits a cero. En caso contrario el valor devuelto sería 1024, que en hexadecimal serían 04 00.

Ahora el maestro tiene que escribir en el esclavo el valor que va a tomar la salida para el led. Como se explicó anteriormente, este valor depende de la salida del sensor PIR, por lo que el valor que escribe el maestro se corresponde con el que acaba de leer en el ejemplo anterior, con la salvedad de que estará mapeado al rango 0-255. Suponemos por tanto que el valor a escribir es cero, ya que el sensor PIR no detectaba movimiento. La trama que envía el maestro para escribir en el esclavo tendría la siguiente forma:

Dirección esclavo	Código de función	Dirección del registro	Valor a escribir	CRC
01	06	00 00	00 00	----

Como se ha comentado anteriormente, en caso de que el sensor PIR mostrase actividad, el valor que habría que escribir sería 255, en hexadecimal 00 FF.

La respuesta del esclavo sería un eco del mensaje en caso de no haber errores.

Segundo escenario: maestro-esclavos

Como ya puede intuir el lector, la naturaleza de los mensajes que se intercambian es exactamente la misma, por lo que también lo serán las tramas. La única diferencia estará en el campo de dirección de esclavo: en este caso podrán tomar los valores 01 o 02 según cual sea el destino del mensaje.

5 CONCLUSIONES Y LÍNEAS DE FUTURO

5.1 Conclusiones

En este proyecto se trataba de hacer un estudio del protocolo MODBUS para después montar una red MODBUS RTU maestro-esclavos. Para ello se buscaron los dispositivos comerciales más adecuados para su integración en el proyecto; se eligieron las placas Arduino UNO con varios módulos de expansión que amplían su funcionalidad y tras instalar todos los componentes a nivel software, programarlos, conectarlos a nivel hardware, identificar y depurar errores se ha conseguido la comunicación entre el maestro y los esclavos por lo que podemos dar por satisfecho el objeto de este trabajo.

Este documento trata de explicar de la manera más clara posible el funcionamiento del protocolo MODBUS, de manera que posibles alumnos interesados en el tema puedan entender el protocolo y realizar el montaje de la red siguiendo los pasos aquí descritos.

Como apunte personal me dirijo a los lectores de este texto para animarlos en sus proyectos futuros, pues es tarea de cualquier ingeniero comprender cómo funciona una tecnología que se desconoce, para después ponerla en práctica. Justo eso es lo que se ha hecho en este proyecto.

5.2 Líneas de futuro

Como posibles líneas futuras se contempla la posibilidad de mejorar la línea de transmisión física que constituye el bus. Para grandes distancias sería interesante la creación de un bus de par trenzado con fichas para las derivaciones a las que se conectarían los nodos de la red. Además, conviene aclarar que según el fabricante habría que desoldar una resistencia de terminación de tipo SMD de 120 ohmios que llevan todos los módulos MODBUS/RS-485 en los nodos intermedios puesto que el fabricante ha diseñado el módulo pensando en que se usaría para dispositivos finales. En nuestro caso, dada la corta distancia a la que se estaba trabajando no ha sido necesario.

Por otra parte sería interesante también implementar una interfaz gráfica con la que poder consultar el estado de los sensores y actuadores vía web. Para ello habría que añadir otro módulo al maestro con el que poder conectarlo a internet y enviar los datos a un servidor que ejecute un software SCADA como puede ser Mango M2M que nos permita acceder a la información desde cualquier lugar con conexión a internet.

Por último, quedan totalmente abiertas a la imaginación otras aplicaciones prácticas que se podrían dar a la red con tan sólo cambiar los sensores. Algunas ideas serían por ejemplo: controlar la temperatura y la presencia en una vivienda, monitorizar un huerto inteligente y activar su regadío o controlar el estado de unas alarmas anti-incendio.

BIBLIOGRAFÍA

[1] http://www.modbus.org/specs.php.

[2] https://www.arduino.cc/.

[3] https://es.wikipedia.org/wiki/Arduino.

[4] https://www.luisllamas.es/detector-de-movimiento-con-arduino-y-sensor-pir/.

[5] https://www.cooking-hacks.com/forum/

Código del maestro:

```
#include <Servo.h>
#include <arduinoUtils.h>

// Librerías para RS-485/Modbus
#include <arduinoRS485.h>
#include <ModbusMaster485.h>

// Instanciamos el objeto ModbusMaster  como esclavo con ID 1
ModbusMaster485 node1(1);
ModbusMaster485 node2(2);



// Definimos las direcciones para leer y escribir en el esclavo
#define address_0  0  //escritura: registro para encender el led en el esclavo2 y registro de escritura  para manejar el ángulo de giro del servo en el esclavo1
#define address_1 1 //registro de lectura sensor PIR del esclavo1
// Definimos el número de registros que se van a leer
#define numRegs 1

int leidoPir;
int angulo=90;

void setup()
{
  // Power on the USB for viewing data in the serial monitor
  Serial.begin(115200);
  delay(100);
  // Initialize Modbus communication baud rate
  // Only allowed in SOCKET1
  node1.begin(74880);
  node2.begin(74880);
```

```
  // Print hello message
  Serial.println("Modbus communication over RS-485");
  delay(100);
}



void loop()
{
  // This variable will store the result of the communication
  // result = 0 : no errors
  // result = 1 : error occurred
//delay(500);
int  result = node1.readHoldingRegisters(address_1, numRegs);
//delay(500);
 if (result != 0) {
    // If no response from the slave, print an error message
    Serial.println("Communication error");
    delay(50);
  }
  else {
    leidoPir = node1.getResponseBuffer(0);
    delay(1000);
    // If all OK
    if(leidoPir>900){
    Serial.println("¡Movimiento detectado!  ");
    }
    delay(100);
  }


      result =  node2.writeSingleRegister(address_0, leidoPir);
  if (result != 0) {
    // Si el esclavo no responde se avisa del error
    Serial.println("Error de comunicacion");
    delay(50);
  }
  else {
    // Si todo OK
    if(leidoPir>900){
```

```
Serial.println("Orden led esclavo2 enviada.  ");
      angulo = 90;
      result =  node1.writeSingleRegister(address_0, angulo);
      result =  node1.writeSingleRegister(address_0, angulo);
      if (result != 0) {
        // Si el esclavo no responde se avisa del error
        Serial.println("Error de comunicacion");
        delay(50);
      }
      else {
        // Si todo OK
        Serial.println("Orden servo esclavo1 enviada: girar 90 grados.  ");
      }
    }
    if (leidoPir < 900)
    {
      angulo = -90;
      result =  node1.writeSingleRegister(address_0, angulo);
      result =  node1.writeSingleRegister(address_0, angulo);
      if (result != 0) {
        // Si el esclavo no responde se avisa del error
        Serial.println("Error de comunicacion");
        delay(50);
      }
      else {

        // Si todo OK
        Serial.println("Sin movimientos.");
        Serial.println("Orden servo esclavo1 enviada: girar -90 grados.  ");
      }
    }
    delay(100);
  }
  Serial.print("\n");
  delay(20);
  // Clear the response buffer
  node1.clearResponseBuffer();
  node2.clearResponseBuffer();
}
```

Código del esclavo1:

```
#include <Servo.h>
#include <arduinoUtils.h>

// Librerías para RS-485/Modbus
#include <arduinoRS485.h>
#include <ModbusSlave485.h>

// Instanciamos los objetos Modbus y servo
ModbusSlave485 mbs;
Servo myservo;

// Registros del esclavo
enum {
  MB_0,   //Empieza por el Registro 0
  MB_1,
  MB_REGS // Para indicar el tamaño de los registros
};

int regs[MB_REGS];
int angulo;

void setup() {

  myservo.attach(3);
  Serial.begin(115200);

  // Parámetros para configurar Modbus
  // Asignamos el SlaveId
  const unsigned char SLAVE = 1;
  // Baud rate
  const long BAUD = 74880;

  // Configure msb with config settings
  // Only allowed in SOCKET1
  mbs.configure(SLAVE, BAUD);
}
```

```
 // Pasamos los valores de los registros al objeto modbus
  mbs.update(regs, MB_REGS);

  // Leemos las entradas analógicas y las asignamos a los registros modbus

  angulo= regs[MB_0];             //angulo de giro del servo, ordenado por el
maestro
  regs[MB_1]=analogRead(A0);
  if(angulo==90)
  {
    Serial.println("Orden recibida: girando 90 grados.");
  }
  else if(angulo==-90)
  {
    Serial.println("Orden recibida: girando -90 grados.");
  }

  myservo.write(angulo);

  delay(10);
}
```

Código del esclavo2

```
#include <arduinoUtils.h>

// Librerías para RS-485/Modbus
#include <arduinoRS485.h>
#include <ModbusSlave485.h>

// Instanciamos objeto Modbus
ModbusSlave485 mbs;

// Registros del esclavo
enum {
  MB_0,   //Empieza por el Registro 0
  MB_REGS // Para indicar el tamaño de los registros
};

int regs[MB_REGS];
int salidaLed;

void setup() {
  //salida para encender el led
  pinMode(4,OUTPUT);

  Serial.begin(115200);

  // Parámetros para configurar Modbus
  // Asignamos el SlaveId
  const unsigned char SLAVE = 2;
  // Baud rate
  const long BAUD = 74880;

  // Configuramos los parámetros de la comunicación
  mbs.configure(SLAVE, BAUD);
}

void loop()
{
```

```
// Pasamos los valores de los registros al objeto modbus
  mbs.update(regs, MB_REGS);


  // Leemos las entradas analógicas y las asignamos a los registros modbus


  salidaLed= regs[MB_0];         //el valor escrito por el maestro se asigna a
la salidaLed
  if(salidaLed > 900){
  Serial.println("Led encendido");
  }
  else
  {
   salidaLed=0;
   Serial.println("Led apagado");
  }
  digitalWrite(4,salidaLed);
  delay(10);
}
```