ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

PROYECTO FIN DE CARRERA

Protocolos de Comunicación en Automatización Industrial

AUTOR: Miguel Cerezuela Martínez
DIRECTOR: José María Malgosa Sanahuja
09/2014

ÍNDICE

CAPÍTULO 1. Sistemas de Automatización Industrial

1 Introducción: Breve Definición de Autómata

"Un autómata programable (PLC) es un equipo electrónico diseñado para ser utilizado en un entorno industrial y destinado al control de procesos industriales con un hardware independiente del proceso a controlar. Dicho hardware se adapta al proceso mediante un programa (software), que contiene las instrucciones a realizar. Esta secuencia de operaciones se define sobre una serie de entradas-salidas cableadas directamente al autómata y con las que este interactúa con el proceso". [1]

1.1Autómatas Programables

1.1.1 Estructura de un Autómata Programable

Es cierto que hoy en día existen infinidad de equipos en el mercado de distintos fabricantes; pero, en cuanto a su arquitectura, podemos afirmar que un autómata consta, esencialmente, de los siguientes bloques:

- Unidad central de proceso (CPU)
- Memorias internas
- Memoria de programa
- Interfaces de entrada salida
- Fuente de alimentación

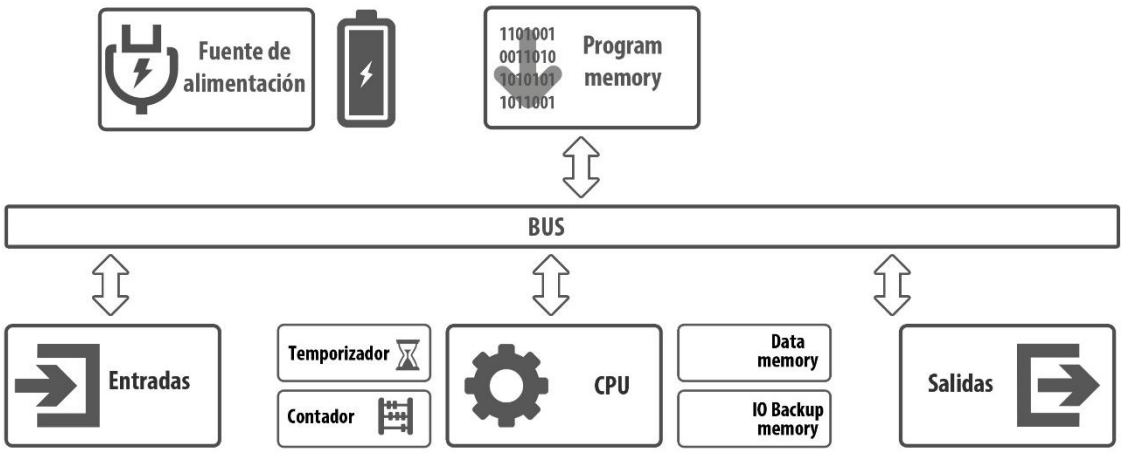

Figura 1. Arquitectura interna de un autómata programable.

[1] Definición IEC 61131

CPU: Es el elemento basado en un microprocesador que ejecuta el programa, realiza la transferencia a las entradas/salidas y, en ocasiones, establece comunicación con dispositivos periféricos, PC u otros autómatas.

La ejecución del programa se lleva a cabo mediante la adquisición por parte de la CPU de las instrucciones almacenadas en la memoria del programa. Una vez transferida una instrucción, se realiza la operación que en ella esta descrita y se pasa a la siguiente. La decodificación de las instrucciones sigue, normalmente, un esquema de lógica procesador-memoria; pero en autómatas de gamas altas la lógica puede estar microprogramada por hardware (lógica cableada).

- **Memorias internas:** Son las encargadas de almacenar el estado de las variables, los valores intermedios de operaciones, el estado de contadores o temporizadores y un reflejo de las entradas y salidas tras la última ejecución del programa. La zona de memoria encargada de esto último se llama memoria de imagen de entradas y salidas.

- **Memoria de programa:** En esta memoria se almacenan el programa escrito por el usuario y algunos parámetros del sistema, como la configuración de las entradas y salidas o las posibles redes de comunicaciones, en caso de que existan.

- **Interfaces de entrada salida:** Establecen la comunicación entre la unidad central y la planta, filtrando, adaptando y codificando, de forma comprensible para dicha unidad, las señales procedentes de los elementos conectados a las entradas. Asimismo, decodifican y amplifican las señales generadas durante la ejecución del programa antes de enviarlas a los elementos conectados a las salidas.

- **Fuente de alimentación:** Es la encargada de generar, a partir de la tensión exterior, todas las tensiones que el autómata necesita para su funcionamiento. Normalmente el autómata cuenta, además, con una batería para alimentar las memorias y poder mantener, en caso de fallo de la tensión de alimentación exterior, el estado de sus entradas/salidas.

1.1.2 Ciclo de Funcionamiento de un Autómata

Los autómatas programables son máquinas que ejecutan las instrucciones de sus programas de forma secuencial, generan las señales de salida dirigidas al proceso a partir de las señales de entrada que reciben del mismo. Esta secuencia puede dividirse en tres fases:

- Lectura de las señales de entrada

- Escritura de señales de salida

- Procesado o ejecución del programa para la obtención de las señales de salida

1.1.3 Modos de operación de un PLC

En los autómatas programables se distinguen los siguientes modos de operación:

- *Modo RUN*: Es el modo de funcionamiento normal según el cual se ejecuta el programa del usuario.

- *Modo STOP*: Es el modo vigente cuando el usuario decide detener la ejecución del programa en el autómata. Las salidas pasan a estado OFF, los registros internos de memoria, así como los contadores y temporizadores, mantienen su valor; cuando pasan de nuevo a modo RUN toman el valor OFF –exceptuando aquellas zonas de la memoria protegidas contra pérdidas de tensión–.

- *Modo ERROR*: El dispositivo entra en este estado cuando se produce un error durante la ejecución del programa de usuario. Sale de él cuando el error se corrige. En el momento en el que el autómata entra en este estado sus salidas quedan en OFF. Puede volver a RUN por distintas razones: reset de la alimentación exterior, decisión de la CPU, un comando enviado desde la unidad de programación por parte del usuario, etc.

El modo STOP se utiliza normalmente para labores de mantenimiento, ya que el funcionamiento queda congelado. Esto facilita tanto la detección de posibles problemas en la programación como la correcta visualización del sistema, en el caso de que se quiera añadir funcionalidades o modificar algún elemento del programa.

Todos los autómatas de gama media-alta tienen la posibilidad de cambiar de modo mediante un conmutador o a través de la unidad de programación; sin embargo, los compactos o de gamas bajas no suelen contar con conmutadores.

1.2 Sensores y Actuadores.

"Se utiliza (...) la palabra sensor para definir al dispositivo o elemento que convierte una variable física no eléctrica en otra eléctrica, que en alguno de sus parámetros (nivel de tensión, nivel de corriente, frecuencia, etc.), contiene información correspondiente a la primera".[2]

Para que en un entorno industrial el sistema de automatización pueda funcionar necesita conocer el valor de numerosas variables físicas que en su gran mayoría no son eléctricas. Es necesario, por lo tanto, la utilización de elementos que conviertan dichas variables en señales eléctricas proporcionales en alguna de sus características, como tensión, corriente o frecuencia.

1.2.1 Clasificación de Sensores:

Son muchas las variables susceptibles de ser transformadas en señales eléctricas. A continuación se presenta una clasificación de acuerdo a un conjunto de características diferentes y no excluyentes.

1.2.1.1 Clasificación según su Principio de Funcionamiento

Según su principio de funcionamiento, podemos distinguir entre sensores generadores (activos) y sensores moduladores (pasivos).

- **Sensores activos:** Son aquellos en los cuales la magnitud física a medir proporciona energía suficiente para generar la señal eléctrica.

2 Mandado y otros, p. 429

- **Sensores pasivos:** Son aquellos en los que la magnitud física a medir modifica alguno de sus parámetros eléctricos, como resistencia o capacidad. Se caracterizan por necesitar alimentación externa.

1.2.1.2 Clasificación según el Tipo de Señal Eléctrica que Generan

Otra característica según la cual se clasifican los sensores es el tipo de señal que generan. Se puede distinguir entre sensores analógicos y digitales.

- **Sensores analógicos**: Son aquellos sensores que generan señales que pueden tomar cualquier valor dentro de un intervalo y cuya amplitud codifica la información generada.

- **Sensores digitales**: Son aquellos que entregan señales, con valores de posibles estados finitos, que codifican el valor medido bien en un solo bits, o en una secuencia de estos.

1.2.1.3 Clasificación de los sensores según el rango de valores

Según el rango de valores que se obtenga a la salida un sensor puede clasificarse como:

- **Sensores de medida:** Son aquellos que proporcionan a su salida todos los valores correspondientes a cada valor de entrada.

- **Sensores todo-nada:** Se caracterizan por distinguir solamente si la variable de entrada está por encima o por debajo de un determinado valor.

1.2.2 Características eléctricas de los sensores analógicos:

Este tipo de sensores entregan su información codificada en tensión o en corriente. En el caso de la codificación por tensión, los valores más habituales suelen ser de 0 a 10 V, de 1 a 5V, de -5 a +5 V, y de -10 a + 10 V. Esta forma de transmitir es más adecuada cuando la longitud de los cables que unen el sensor con el dispositivo de medida es corta; esto es debido a que la caída de tensión en el cable es proporcional a su longitud.

Los sensores que utilizan la corriente como parámetro para transmitir información no sufren el inconveniente de la caída de tensión en el cable, puesto que en un circuito serie discurre la misma corriente en todos sus puntos. Este tipo de sensores entregan comúnmente los siguientes rangos de valores de corriente:-20 a +20mA, 4 a 20mAy 0 a 20mA.

La mayoría de los autómatas programables del mercado incluyen entradas y salidas o módulos de ampliación capaces de trabajar con estos valores de tensión o corriente.

1.2.3 Características eléctricas de los sensores digitales.

En los sensores de salida digital la corriente de carga es un parámetro fundamental. Esta corriente depende de la tensión en la salida. Esta salida puede tener dos configuraciones diferentes:

- **Salida con transistor NPN y resistencia de carga:** Proporciona niveles de tensión y de corriente compatibles con las dos familias lógicas de uso más extendido (CMOS y TTL).

- **Salida con transistor NPN y colector abierto:** Esta configuración no tiene incorporado la resistencia R de la **Figura 2**. Colocando una resistencia externa del valor adecuado, puede adaptarse la señal a TTL y CMOS.

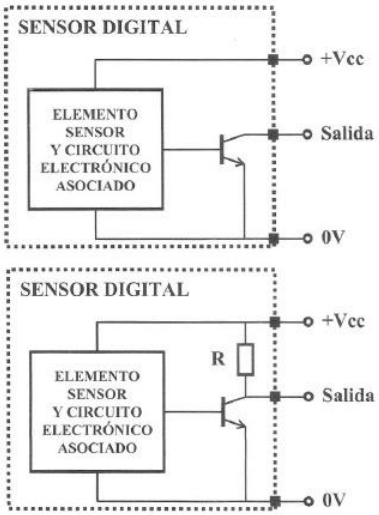

Figura 2. Posibles configuraciones de una salida digital.

1.3. Interfaces hombre-máquina y sistemas de adquisición de datos.

Son sistemas que permiten al operador de una planta interactuar con el proceso. Una interfaz hombre-máquina (HMI) es una abstracción de la forma en que el usuario interactúa con una máquina y puede ser desde una botonera en un cuadro eléctrico hasta una pantalla táctil comunicada con uno o varios autómatas programables a través de una red de comunicación industrial.

Un sistema de supervisión y adquisición de datos (SCADA) es una aplicación informática que se comunica con los elementos de campo de una planta para obtener y procesar los datos del proceso y mostrarlos al operador de una forma intuitiva y comprensible. Además también permite al operador operar sobre las máquinas por lo que podríamos decir que un HMI es un subconjunto de un los sistemas SCADA.

Un SCADA consta de las siguientes partes:

- Una base de datos que permite guardar los parámetros de configuración de las variables que tomaran los valores obtenidos del proceso y a las comunicaciones con los dispositivos e historiar dichas variables.

- Un software de gestión de alarmas que permita al operador localizar de forma rápida las averías que se produzcan en el proceso.

- Un software que implemente un driver de comunicaciones para enviar y recibir información hacia el proceso.

Con respecto a las comunicaciones, debido a que cada fabricante de dispositivos de automatización industrial emplea un protocolo diferente, para cada sistema SCADA existen multitud de drivers que permiten intercambiar información con autómatas y dispositivos de múltiples compañías.

1.3.1 Algunos ejemplos de Interfaces hombre-máquina y SCADA.

En la actualidad existen en el mercado incontables aplicaciones SCADA, tanto soluciones comerciales como de software libre, mientras que en el caso de os HMI al implicar un hardware más o menos complejo, el número de productos es menos y se restringe a unos pocos fabricantes de dispositivos de automatización industrial como Omron o Schneider Electric.

1.3.1.1 Ejemplos de HMI.

Hoy en día existen infinidad de HMIs en el mercado. A diferencia de los SCADA en los que mediante un driver es posible comunicarse con dispositivos de múltiples fabricantes, las interfaces hombre-máquina suelen estar preparadas para obtener datos de dispositivos de su mismo fabricante.

Lo dispositivos más utilizados en la actualidad son las pantallas táctiles. Estas pantallas están preparadas para ser montadas directamente en los cuadros eléctricos de control y se configuran y programan mediante un software que proporciona el fabricante. Aunque como se ha dicho en el anterior apartado, una HMI puede ser una botonera convencional de un cuadro eléctrico, la versatilidad de las pantallas táctiles programables hace que sean una opción más barata a largo plazo, puesto que si el sistema de automatización cambia estas pueden adaptarse de forma sencilla.

Figura 3. Gama Magelis de HMIs de Schneider Electric.

1.3.1.2 Ejemplos de SCADA.

En los últimos años se ha producido un gran avance cualitativo en el mundo de los SCADA motivado por la aparición de algunas aplicaciones que han trasladado conceptos más propios de la programación de dispositivos móviles, aplicaciones web, o aplicaciones distribuidas al campo de la automatización industrial.

Un buen ejemplo es Ignition de la compañía norte americana Inductive Automation. Este software está programado en Java puro, por lo que puede utilizarse con cualquier sistema operativo que soporte este lenguaje de programación. Su arquitectura permite ejecutar la aplicación en un servidor y lanzarla vía web en múltiples clientes desde los cuales además puede editarse. Ignition cuenta con drivers de comunicaciones para prácticamente todos los protocolos de comunicación industrial.

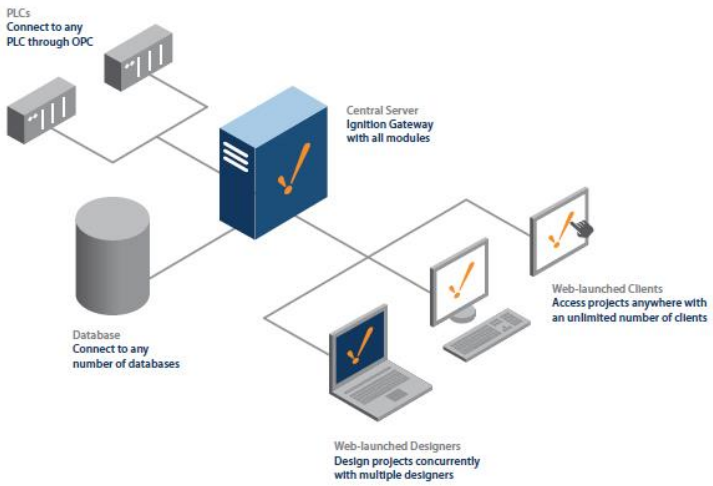

Figura 4. Arquitectura básica del SCADA Ignition de Inductive Automation.

Otras características muy interesantes de este software son: su sencilla conexión con bases de datos, la posibilidad de crear arquitecturas redundantes y la existencia de un SDK que permite añadirle funcionalidades programadas en Java por el usuario.

CAPÍTULO 2. Protocolos de comunicación y buses de campo en entornos industriales

2 Introducción

A finales de los años sesenta, el elevado coste de los sistemas de automatización basados en relés hizo evidente la necesidad de crear una nueva tecnología que permitiese modificaciones en el sistema que controla el proceso de producción, sin realizar cambios en el cableado. Con este objetivo la empresa Bedford Associates, propuso a General Motors, conocida compañía del sector automovilístico, el MODICON 084: el primer PLC de la historia.

Alrededor del año 1973, una vez que la tecnología había madurado lo suficiente, fue posible la creación de un sistema que permite la comunicación recíproca de los PLC's. Este sistema se conoce como MODBUS y ofrece la posibilidad de comunicar dos equipos siguiendo un esquema maestro-esclavo.

Actualmente existe un grupo de empresas distribuidoras de tecnología para la automatización industrial –integradoras o desarrolladoras de la misma–, que ofrece soporte para el empleo o desarrollo de dispositivo que utilizan el protocolo MODBUS.

2.1 Modbus

Modbus es uno de los protocolos de comunicación industrial más utilizados en la actualidad debido a que por su sencillez y su carácter abierto, su uso está ampliamente extendido entre múltiples fabricantes de dispositivos. A continuación se exponen las características más importantes del protocolo así como las diferentes opciones existentes a la hora de implementarlo.

2.1.1 Arquitectura Protocolar

Modbus en un protocolo que ocupa el nivel 7 (aplicación) dentro del modelo de referencia OSI. Se implementa de las siguientes tres formas:

- TCP: Utilizando Ethernet como enlace de datos y acceso al medio.
- Transmisión serie asíncrona sobre diversos medios: RS232/422/485.
- Modbus: Red de alta velocidad con paso de testigo. Utiliza HDLC como nivel de enlace de datos.

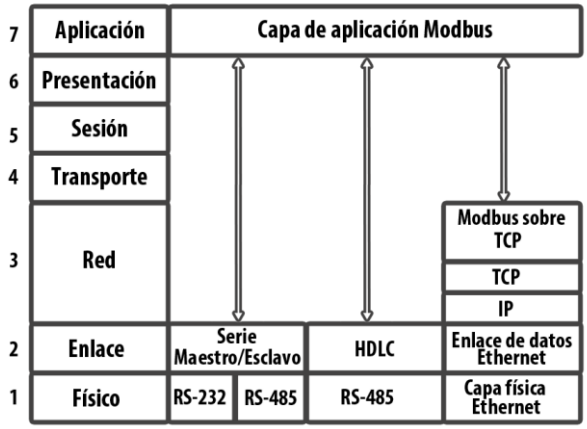

Figura 5. Arquitectura de Modbus según el modelo OSI.

2.1.2 Formato de las Tramas

El protocolo Modbus define una unidad de datos de protocolo (PDU) muy sencilla y totalmente independiente de las capas de comunicación adyacentes. El mapeo de protocolo MODBUS, en buses o redes específicos, puede introducir algunos campos adicionales en la unidad de datos de la aplicación (ADU).

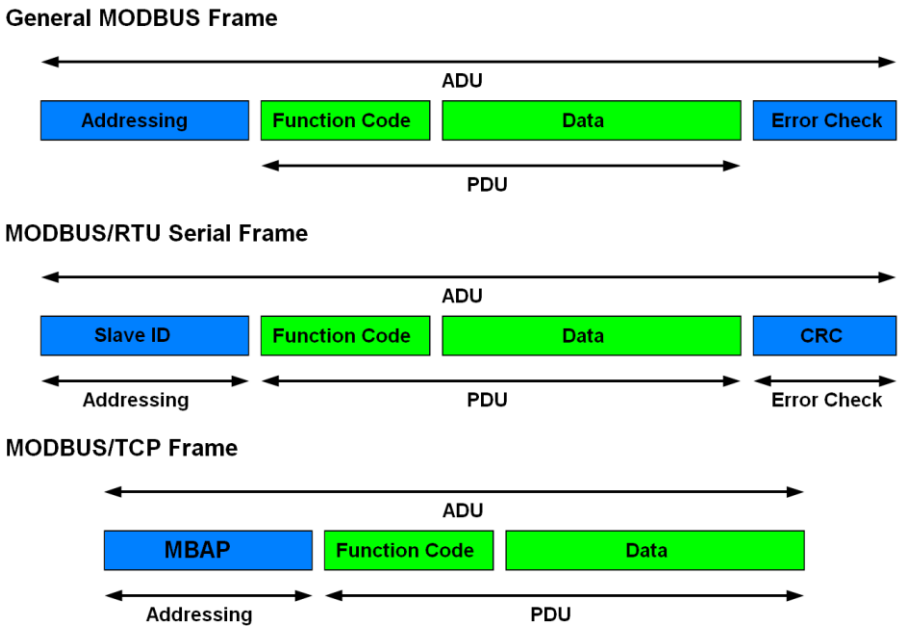

Figura 6: Formato de las tramas Modbus.

PDU: Es la unidad de datos básica en el protocolo Modbus. Consta de dos campos:

- *Código de Función*: codifica el tipo de acción a realizar por parte del servidor. Ocupa un Byte.
- *Campo de Datos*: tiene que ver con el mensaje. Si este ha sido enviado por el cliente hacia el servidor, contendrá información que el servidor necesita para ejecutar la acción indicada por el código de función. Si, por el contrario, el mensaje ha sido enviado por el servidor podrá contener, bien los datos solicitados por el cliente, bien un código de error –que indicará que la acción solicitada no se ha podido llevar a cabo y la causa–.

ADU: Es la unidad de datos del protocolo en la práctica. Tiene un campo adicional que depende del tipo de red o bus sobre el que se use el protocolo. En el caso de Modbus RTU este campo ocupa un Byte y se emplea para identificar al esclavo origen o destino de los datos. Si se emplea Modbus TCP este campo es algo más complejo, por lo que se detallará con mayor profundidad en el apartado dedicado a esta versión del protocolo.

Los mensajes de tipo **petición,** que viajan desde esclavo hacia el maestro, están compuestos por el código de función y los datos. La composición de los datos depende del tipo de petición, que puede ser de lectura o escritura:

- **Lectura**. En el caso expuesto el campo de datos está compuesto por una *dirección de inicio* –a partir de la cual leer– y un *número de datos* a leer. Ambos campos tienen una longitud de dos Bytes.

Código de función (1 Bytes)	Dirección (2 Bytes)	Numero de datos (2Bytes)

- **Escritura**. El campo de datos tiene diferentes estructuras según el tipo y el número de datos que se desee escribir.

Escritura simple de valor binario:

Código de función (1 Byte)	Dirección (2 Bytes)	Datos a escribir (2Bytes)

Escritura múltiple de valores binarios:

Código de función (1Byte)	Dirección (2 Bytes)	Numero de valores (2 Bytes)	Número de bytes =**N*** (1 Byte)	Datos a escribir (**N*** Bytes)

Escritura simple de registros:

Código de función (2 Bytes)	Dirección (2 Bytes)	Datos a escribir (2Bytes)

Escritura múltiple de registros:

Código de función (1Byte)	Dirección (2 Bytes)	Numero de valores (2 Bytes)	Número de byte (1 Byte)=**N**	Datos a escribir (2x**N** Bytes)

El tamaño de la PDU MODBUS está limitado por la restricción de tamaño, heredado de la primera aplicación MODBUS en red de línea serie (máx. RS485 ADU = 256 bytes).

Por lo tanto, MODBUS **PDU comunicación serie =** 256 – Dirección servidor (1 byte) - CRC (2 bytes) = 253 bytes, y en consecuencia:

- RS232 / RS485 **ADU** = 253 bytes + Server address (1 byte) + CRC (2 bytes) = **256 bytes**
- TCP MODBUS **ADU** = 253 bytes + MBAP (7 bytes) = **260 bytes**.

Con respecto a la **codificación de los datos**, Modbus usa una representación big-endian para direcciones e ítems de datos. Esto significa que cuando se transmite un dato más largo de un byte, el byte más significativo es enviado el primero.

2.1.3 Modelo de Datos

El modelo de datos en Modbus distingue entre estradas digitales, salidas digitales (coils), registros de entrada y registros de retención (holding registers). Las entradas y salidas digitales ocupan, evidentemente, un bit; mientras que los registros, tanto de entrada como de retención, ocupan dos Bytes.

Primary tables	Object type	Type of
Discretes Input	Single bit	Read-Only
Coils	Single bit	Read-Write
Input Registers	16-bit word	Read-Only
Holding Registers	16-bit word	Read-Write

Figura 7. Modelo de datos en el protocolo Modbus.

Para cada una de los tipos de datos, el protocolo permite la selección individual de 65.536 elementos. Las operaciones de lectura o escritura de estos están diseñadas para abarcar varios elementos de datos consecutivos hasta un límite de tamaño de datos que es dependiente del código de funció

2.1.4 Modelo de Direcciones

Todos los datos que se manejan a través de MODBUS han de estar en la memoria del dispositivo, pero es muy importante no confundir la dirección física con la referencia de los datos. Esto significa que el dispositivo, en su implementación del protocolo, debe relacionar las direcciones físicas de los datos con referencias Modbus válidas. Dichas referencias son índices enteros a partir de cero y su especificación depende, exclusivamente, del fabricante del dispositivo.

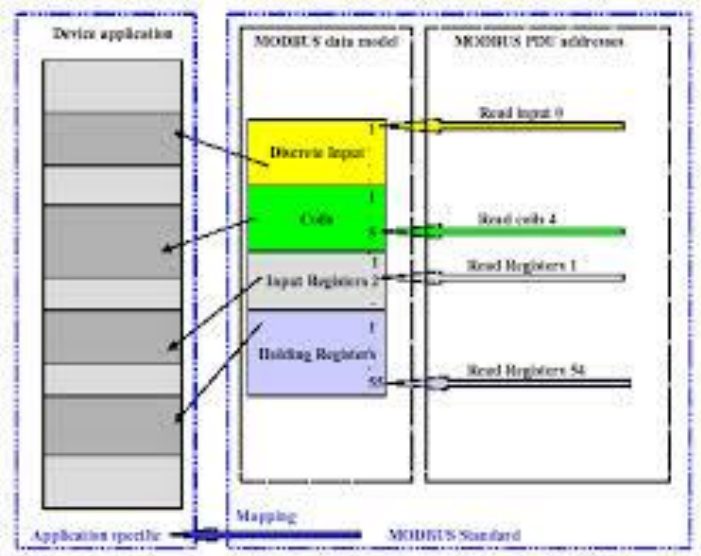

Figura 2. Moelo de direcciones Modbus.

2.1.5 Intercambio de Mensajes

Como se ha expuesto en los apartados anteriores, existen tres tipos básicos de mensajes en el protocolo Modbus: solicitud, respuesta y error. A continuación se detalla cómo se produce el intercambio de dichos mensajes y el formato con el que se construyen según la función del protocolo que se emplee.

2.1.6 Definición de Transacción Modbus

El siguiente diagrama muestra como el servidor procesa la petición respondiendo con el mismo código de función, si la operación ha sido satisfactoria, o con el código de función de error (0X08) –seguido de un código de excepción que codifica el tipo de problema que se ha producido–.

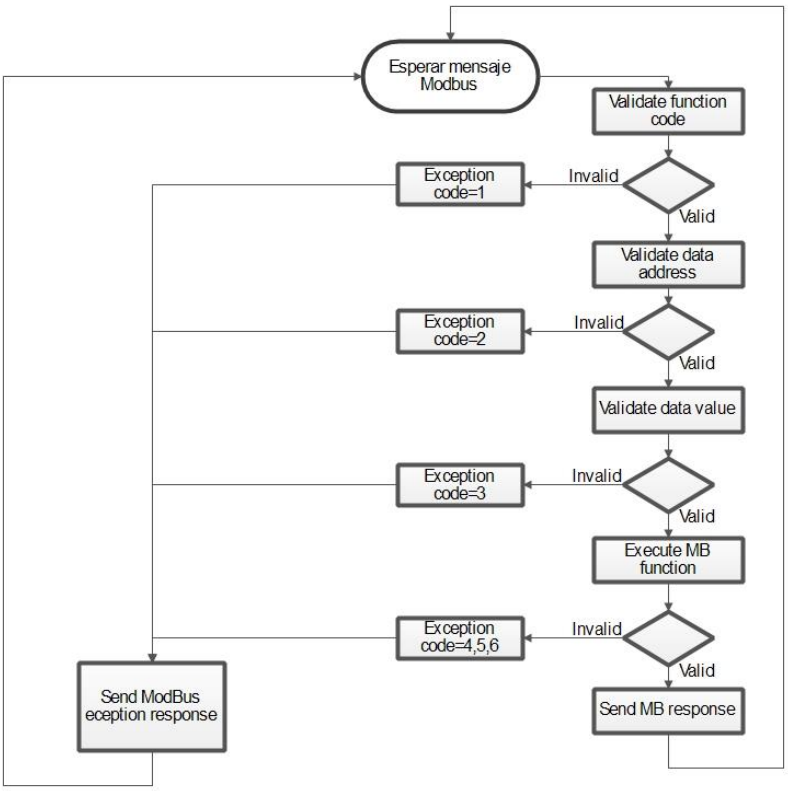

Figura 8. Diagrama de flujo de una transacción Modbus en el lado del servidor.

2.1.7 Códigos de Función Modbus

Los códigos de función del protocolo Modbus se dividen en tres categorías:

- *Códigos de función públicos:* Estos códigos están completamente definidos, tienen la garantía de ser únicos; están validados y documentados por la comunidad MODBUS.org e incluyen una serie de códigos reservados para un posible futuro uso.

- *Códigos de función definidos por el usuario:* Existen dos rangos de códigos de función definibles por el usuario, de forma que es posible definir e implantar funciones que no están soportadas por las especificaciones. No existen garantías de la nueva función definida sea única.

- *Códigos de función reservados:* Son códigos utilizados por algunas empresas para productos antiguos y que no están disponibles para el uso público.

Bit acces	Entradas físicas discretas	Lectura de entradas discretas	02
	Bits internos o salidas físicas	Lectura de salidas discretas	01
		Escritura de un bit de salida	05
		Escritura de multiples salidas	15
16 Bit access	Regístros físicos de entrada	Lectura de registro de entrada	04
		Lectura de registro de retención	03
	Registros internos o registros de salida físicos	Escritura de un registro	06
		Escritura de míltiples registros	16
		Lectura/escritura de múltiples registros	23
Acceso a ficheros		Leer fichero	20
		Escribir ficheros	21

Figura 9. Códigos de función Modbus-

A continuación se exponen los códigos de función más utilizados con ejemplos de su utilización y de la configuración de los mensajes a los que dan lugar.

2.1.8 Posibilidades de Implementación

Debido a su sencillez y al hecho de que Modbus es un protocolo abierto y muy bien documentado, existen múltiples posibilidades a la hora de crear dispositivos hardware e implementar el protocolo sobre ellos.

A continuación se detallan las posibilidades a nivel Hardware, así como las distintas librerías software disponibles para implementar Modbus:

2.1.8.1 Implementación de Modbus en un PC

Esta es la opción más sencilla si queremos implementar Modbus, ya que contamos con un sistema operativo que facilita mucho la realización de algunas tareas –como acceder al hardware de comunicaciones–. Además ofrece la posibilidad de crear un entorno gráfico para interactuar con el usuario final. Por otro lado, la complejidad de los PC's hace que no sean la solución más estable y fiable, por lo que se descarta para la creación de aplicaciones de control de planta. Sin embargo, es la mejor elección para aplicaciones de supervisión de procesos como SCADA.

JAMOD: Es un proyecto que representa una implementación 100% Java del protocolo. Permite crear maestros y esclavos Modbus, tanto en su versión RTU (serie) como TCP/IP, con una API orientada a objetos, comprensible, bien documentada, fácilmente reutilizable y extensible. Además es libre y de Código abierto.

Al estar programada en Java puede ser ejecutada en cualquier sistema operativo que sea capaz de correr este tipo de aplicaciones (OSX, Linux, Windows).

WSMBT es un sencillo componente .net que permite obtener datos de un esclavo ModbusTCP mediante VB.NET, C# o C++.

Libmodbus en una librería de software libre que permite enviar y recibir datos mediante el protocolo Modbus. Está desarrollada en C y permite utilizar las versiones RTU y TCP. Puede compilarse para Linux, FreeBSD, MACOSX y Windows. Al ser una solución compleja puede ser más eficiente en aplicaciones grandes, pero su utilización es más complicada que otras librerías como Jamod y además no es multiplataforma.

2.1.8.2 Implementación de Modbus en un Mini-PC.

En los últimos años están proliferando multitud de mini-PC's, basados sobre todo en procesadores ARM, que representan una muy buena opción para ciertas aplicaciones. En la actualidad estos dispositivos poseen memoria RAM y un procesador suficientemente potente como para correr sistemas operativos como Linux. Además, la facilidad de acceso al conexionado del hardware hace muy sencillo la integración de sensores, actuadores y periféricos.

Un buen ejemplo de este tipo de dispositivos es la **Raspberry Pi**. Esta mini-computadora se creó en la Universidad de Cambridge con fines didácticos, pero pronto empezó a ser utilizado por muchas empresas.

El modelo B de la **Figura 10** cuenta con un procesador Broadcom BCM2835, 512 MB de memoria RAM, Ethernet, 3 puertos USB, HDMI, salida de audio y acceso a las entradas y salidas de propósito general, que pueden ser utilizadas con multitud de propósitos. Su tamaño es aproximadamente el de una tarjeta de crédito.

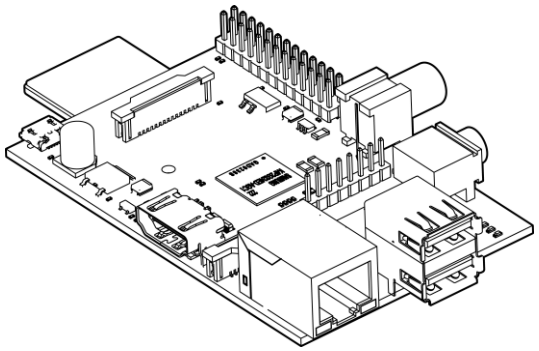

Figura 10. Mini PC Raspberry PI.

Existen varias opciones en lo que a sistema operativo se refiere, pero la más extendida es Raspbian: una distribución Debian compilada para Raspberry y con la que la mayoría de los usuarios de Linux están familiarizados.

Gracias a la gran comunidad de desarrolladores y usuarios que se ha creado en torno a Rapberry, existen numerosos recursos software para implementar todo tipo de servicios. A continuación se muestran algunos de especial interés en la creación de aplicaciones que empleen el protocolo **Modbus**:

- **RPi.GPIO:** Es una librería escrita en Python que proporciona acceso a las entradas y salidas de propósito general.

- **Pymodbus:** Es una implementación integral de Modbus escrita en Python que cuenta con todas las funcionalidades del protocolo en sus versiones RTU y TCP.

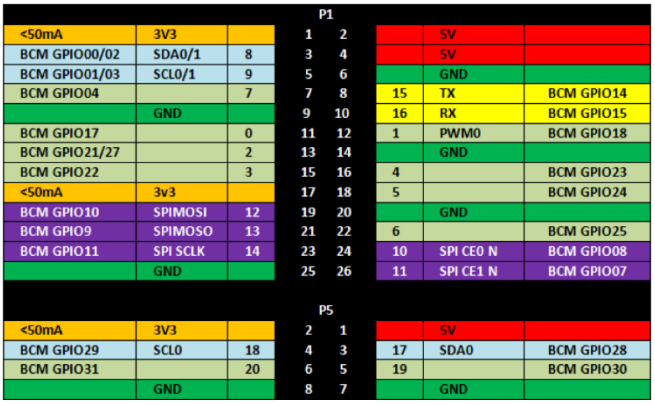

Figura 11. Entrada y salidas de propósito general de la Raspberry PI.

La implementación de Modbus TCP no necesita hardware adicional; sin embargo, para Modbus RTU es necesario algún elemento extra para el acceso al medio físico –lo que no suele suponer ningún inconveniente, pues son sencillos y de bajo coste–.

2.1.8.3 Implementación de Modbus en un Microcontrolador

Los microcontroladores son una buena opción para desarrollar dispositivos industriales que requieran mayor fiabilidad que un PC. Durante la programación de estos elementos el diseñador tiene un control total de la ejecución de las instrucciones y del tiempo en que se ejecutan. Por el contrario, requieren conocimientos avanzados de programación y diseño de circuitos electrónicos.

Uno de los líderes en la fabricación de microcontroladores es **Microchip**, con multitud de productos para el diseño de dispositivos electrónicos orientados a la automatización industrial, así como infinidad de librerías software para el desarrollo de aplicaciones con sus dispositivos.

Existen numerosos dispositivos de prototipado que facilitan el diseño de aplicaciones para entornos industriales.

- **PIC32 Ethernet Starter Kit II:** Este dispositivo cuenta con un microcontrolador de 32 bits, conectores USB, conector RJ45 para conectividad Ethernet y acceso a las entradas/salidas. Es una buena elección para el desarrollo de aplicaciones Modbus en sus variantes TCP y RTU; sin embargo, por su fiabilidad y su escasa capacidad para trabajar con grandes volúmenes de datos, es preferible utilizarla para implementaciones de Modbus RTU, como sensores o actuadores, que poseen una lógica más o menos compleja.

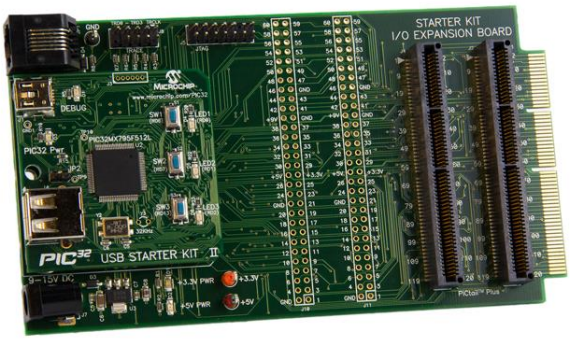

Figura 12. PIC32 Ethernet Starter Kit II.

2.2 Profibus

"Profibus se desarrolló bajo un proyecto financiado por el gobierno alemán. Está normalizado en Alemania por DIN E 19245 y en Europa por EN 50170. El desarrollo y posterior comercialización ha contado con el apoyo deimportantes fabricantes como ABB, AEG, Siemens, Klóckner-Moeller, etc, llegando en la actualidad a ofrecer un elevado número de productos compatibles y exhibiciones conjuntas que demuestran que la red es capaz de integrar y gestionar productos de diferentes marcas bajo un bus de comunicaciones gestionado con un software único. Estác ontrolado por la PNO (Profibus User Organisation) y la PTO (Profibus Trade Organisation).La independencia de los vendedores junto con la eficacia del bus, demostrada en más de 200000 aplicaciones, han provocado un aumento en el porcentaje de mercado industrial en Europa de más del 40% según diversos estudios
indepedientes de mercado". [3]

Existen **tres perfiles** diferenciados:

- **Profibus DP** (Decentralized Periphery). Orientado a sensores/actuadores enlazados a autómatas programables o terminales.

- **Profibus PA** (Process Automation). Para control de proceso y cumpliendo normas especiales de seguridad para la industria química (IEC 1 1 15 8-2, seguridad intrínseca).

- **Profibus FMS** (Fieldbus Message Specification). Para comunicación entre células de proceso o equipos de automatización. La evolución de Profibus hacia la utilización de protocolos TCP/IP para enlace a nivel de proceso hace que este perfil esté perdiendo importancia.

La siguiente tabla muestra las principales características los tres perfiles de Profibus:

[3] Kaschel y Pinto, p. 3

	PROFIBUS-FMS	PROFIBUS-DP	PROFIBUS-PA
Aplicación	Nivel de célula	Nivel de campo	Nivel de campo
Estándar	EN 50 170/IEC 61158	EN 50 170 /IEC 61158	IEC 1158-2
Dispositivos conectables	PLC, PG/PC, Dispositivos de campo	PLC, PG/PC, Dispositivos de campo binarios y analógicos, accionamientos, OPs	Dispositivos de campo para áreas con riesgo de explosión y 31.25 kbit/s
Tiemp. respuest.	< 60 ms	1 - 5 ms	< 60 ms
Tamaño red	<= 150 km	<= 150 km	Máx. 1.9 km
Velocidad	9.6 kbit/s - 12 Mbit/s	9.6 kbit/s - 12 Mbit/s	31.25 kbit/s

Figura 13. Características de los perfiles Profibus.

2.2.1Arquitectura Protocolar

La arquitectura de Profibus está orientada al sistema estandarizado OSI, implementando tres de las siete capas que dicho modelo propone.

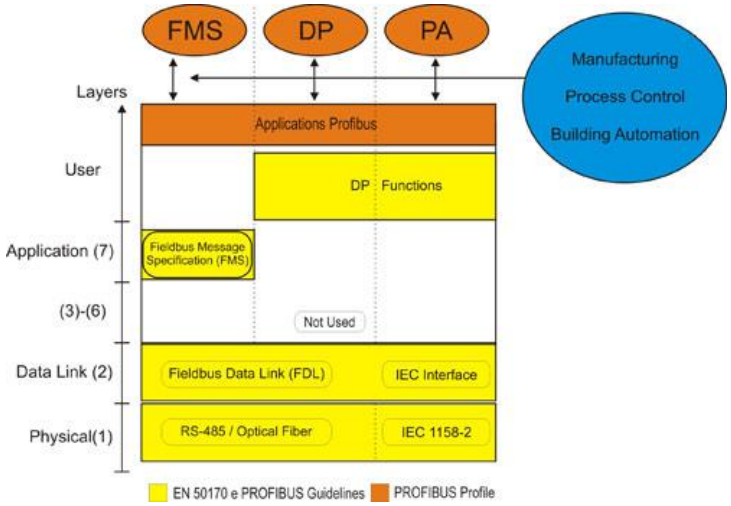

Figura 14. Arquitectura de Profibus según el modelo OSI.

- *La Capa 1* o *Capa física* define características de la transmisión –como el material de medio o los niveles de tensión–, y garantiza la conexión, aunque no de forma fiable.

- *La Capa 2* o *Capa de enlace* (FDL – Fieldbus Data Link) define el protocolo de acceso al bus y se encarga de establecer el orden de circulación del testigo una vez inicializado el bus, adjudicando el testigo en el arranque, en caso de pérdida del mismo o de adición o eliminación de estaciones activas

- *La Capa 7* o *Capa de aplicación* define las funciones de aplicación ofreciendo la posibilidad de acceder a los servicios de las demás capas.

2.2.1.1 Profibus DP

El perfil DP de Profibus utiliza las capas físicas y de enlace del modelo OSI, mientras que no define de las capas 3 a la 7. Permite una comunicación RS-485 o por fibra óptica y asegura una transmisión de datos rápida y eficiente. Las funciones de aplicación y el comportamiento del sistema se especifican en la interfaz de usuario.

Para la capa física Profibus DP se pueden utilizar distintas tecnologías, pero lo más común es RS-485.

"El transporte en Profibus-DP se realiza por medio de tramas según IEC 870-5-1. La comunicación se realiza por medio de datagramas en modo broadcast o multicast. Se utiliza comunicación serie asíncrona por lo que es utilizable una UART genérica. Profibus DP prescinde de los niveles ISO 3 a 6 y la capa de aplicación ofrece una amplia gama de servicios de diagnóstico, seguridad, protecciones etc. Es una capa de aplicación relativamente compleja debido a la necesidad de mantener la integridad en el proceso de paso de testigo (un sólo testigo)".[4]

2.2.1.2 Profibus FMS

Este perfil emplea las capas física, enlace y aplicación. La capa de aplicación está formada por dos subcapas:

- FMS (Fieldbus Message Specification): Es la subcapa de aplicación encargada del protocolo de aplicación, proporcionando al usuario una amplia gama de servicios de comunicación.
- LLI (Lower Layer Interface): Se encarga de proporcionar a FMS un acceso independiente a la capa de enlace.

"Profibus FMS es una compleja capa de aplicación que permite la gestión distribuida de procesos al nivel de relación entre células con posibilidad de acceso a objetos, ejecución remota de procesos etc. Los dispositivos de definen como dispositivos de campo virtuales, cada uno incluye un diccionario de objetos que enumera los objetos de comunicación".[5]

2.2.1.3 Profibus PA

Este perfil Profibus utiliza el protocolo DP extendido, lo que hace que sea posible integrarlo en redes Profibus DP. Para la transmisión de datos usa un indicador que define el comportamiento de los dispositivos de campo. La tecnología de transmisión permite un alto grado de seguridad.

Para la capa física en redes de campo, Profibus PA implementa la norma IEC 11158-2. Esta norma define una comunicación síncrona utilizando modulación sobre la línea de alimentación de los dispositivos. En cambio, a nivel de proceso se suele emplear Ethernet o fibra óptica.

4 José David Jiménez
5 Álvaro López Martínez

2.2.2 Tramas

Las tramas en Profibus admiten los tres tipos de formatos:

- Tramas de longitud fija sin datos.
- Tramas de longitud fija con datos.
- Tramas de longitud variable.

La interpretación de dichas tramas es algo compleja debido a la variedad de tipos previstos para dar servicios a dispositivos con distinto nivel de complejidad, por lo que nos centraremos en explicarlas dentro de los mensajes básicos (cíclicos y acíclicos) que ofrece el protocolo a nivel de enlace.

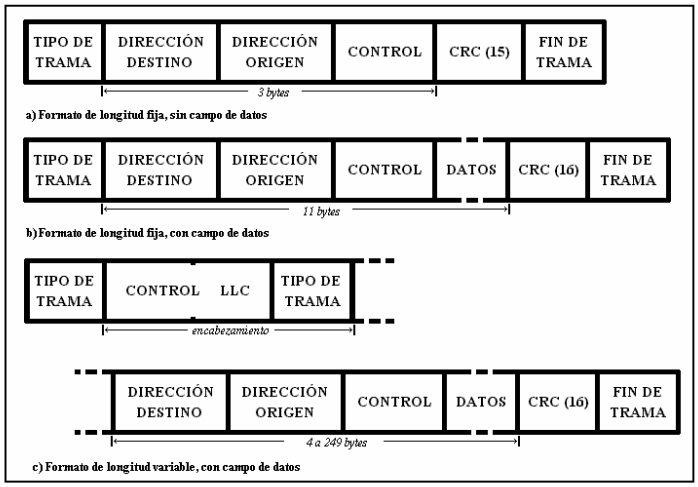

Figura 15. Tramas Profibus.

Profibus es un protocolo capaz de dar servicio a dispositivos con distintos niveles de complejidad, lo que hace que la interpretación de las tramas sea muy compleja, por lo que se explicarán con mayor profundidad en el siguiente apartado, dentro de los mensajes más importantes que maneja el protocolo.

2.2.3 Mensajes intercambiados.

El protocolo objeto de estudio en el presente capítulo contempla dos tipos de mensajes básicos: mensajes cíclicos y acíclicos.

Los mensajes cíclicos permiten el intercambio de datos cuya prioridad, en cuanto al tiempo, es baja. Estos son los tipos de mensajes cíclicos que ofrece Profibus.

- SDN (Send Data with No acknowledge): Son mensajes de difusión del servidor hacia los clientes que no requieren confirmación.

- SDA (Send Data with Acknowledge): Los envía el maestro a uno de los esclavos y contienen datos o funciones de control. Requiere reconocimiento por parte del esclavo.

- RDR (Request Data with Reply): Mensajes punto a punto cuya función es la de solicitar datos a uno de los esclavos.

- SRD (Send and Request Data): Mensajes punto a punto que permiten al maestro enviar datos y recibir datos de un esclavo.

Los mensajes acíclicos permiten acortar el tiempo de respuesta de los datos críticos. A cada turno de Maestro se puede enviar un mensaje de difusión conteniendo los valores críticos de todos los esclavos. La lista de estos valores es conocida por todos maestros. Los mensajes pueden ser de dos tipos:

- CRDR (Cyclic Request Data with Reply)

- CSRD (Cyclic Send and Request Data)

La petición de estos mensajes se realiza mediante un telegrama especial de difusión, que contiene de forma encadenada las peticiones a todos los esclavos.

Figura 16. Telegrama de difusión Profibus.

Las respuestas se producen de forma escalonada mediante una instrucción de lectura rápida en cada uno de los esclavos, pero sin tener que esperar el tiempo de procesamiento de la orden, puesto que la petición se hizo ya anteriormente mediante el mensaje de difusión.

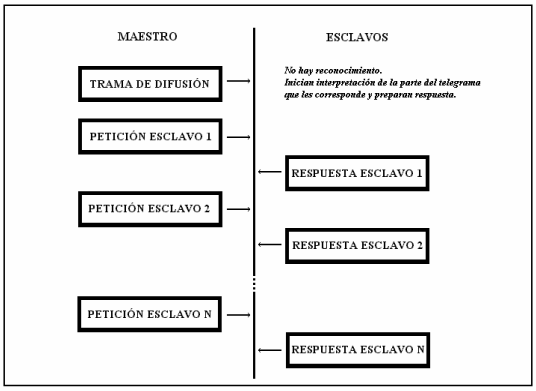

Figura 17. Respuestas a un mensaje de difusión.

"Las tramas de los telegramas admiten, como ya se ha dicho, formatos muy diversos, dependiendo del tipo de aplicación. Dentro de la Organización de usuarios de PROFIBUS se han formado distintos grupos que han desarrollado los detalles de protocolo para distintos campos de aplicación". [6]

6 Profibus (PA/DP/FMS), ETS de Ingenieros Industriales, Universidad Politécnica de Cartagena

2.2.4 Posibilidades de Implementación

Profibus es un protocolo complejo y no es abierto como Modbus. Su implementación es bastante más compleja si se pretende hacer desde cero a partir de un microcontrolador o mini PC. Afortunadamente existen en el mercado multitud de circuitos integrados que implementan el protocolo y que tienen un bajo coste.

2.2.4.1 Implementación de Profibus mediante el uso de chips dedicados.

Existen diferentes circuitos integrados en el mercado para este propósito que pueden ser utilizados junto con microcontroladores o miniPC.

- **Implementación por Chip sencillo para esclavos DP:** Se emplea este método cuando se quiere crear un dispositivo esclavo de campo, tipo sensor o actuador que no posee "inteligencia". Solo es necesario el uso del circuito integrado LSPM2 de Siemens, un director de interface y un reloj como componentes externos.

- **Implementación de esclavos inteligentes FMS y DP:** Las partes críticas en el tiempo del protocolo Profibus son implementadas en un chip protocolar, mientras que la "inteligencia" del dispositivo se programa en un microcontrolador. Existen en el mercado multitud de chips para esclavos inteligentes como el 68360 de Motorola o el SPC3 de Siemens.

- **Implementación de maestros complejos FMS y DP:** Se implementan de la misma forma que los casos explicados anteriormente. También podemos encontrar multitud de dispositivos en el mercado (PBM de IAM, ASPC2 de Siemens).

- **Implementación de dispositivos de campo PA:** Cuando se implementa un dispositivo de campo PA es esencial que este tenga un bajo consumo puesto que normalmente no se permiten corrientes superiores a 10mA. Chips como el SIM 1 de Siemens han sido diseñados cumpliendo este requisito.

2.2.4.2 Implementación de Profibus mediante el uso de mini PCs.

Aunque como se ha expuesto anteriormente el uso de mini PCs no es lo más recomendable a la hora de crear dispositivos que se comuniquen con Profibus, estos pueden utilizarse cuando los requerimientos de tiempo no son muy estrictos o para dar lógica en un dispositivo delegando las comunicaciones en uno de los chips descritos en el apartado anterior.

El proyecto **Profibus on Raspberry Pi** es un proyecto Open hardware y software y representa una implementación tanto de la capa física del protocolo como de las capas de enlace de datos y aplicación, si bien estas dos últimas aún están en fase de desarrollo. Toda la información de este proyecto puede consultarse en bues.ch/cms/hacking/profibus.html.

2.3 Otros Buses de Campo y Protocolos de Automatización Industriales

En este apartado se pretende dar un repaso a otros protocolos de comunicación industrial y buses de campo, que por otra parte no tienen un uso ni mucho menos tan extendido como Modbus o Profibus.

Comparación de características entre algunos buses y protocolos

Nombre	Topología	Soporte	Máx dispositivos	Rate Transm. bps	Distancia máx Km	Comunicación
Profibus DP	linea, estrella yanillo	par trenzado fibra óptica	127/segm	Hasta 1.5M y12M	0.1 segm 24 fibra	Master/Slave peer to peer
Profibus PA	linea, estrella yanillo	par trenzado fibra óptica	14400/segm	31.5K	0.1 segm 24 fibra	Master/Slave peer to peer
Profibus FMS		par trenzado fibra óptica	127/segm	500K		Master/Slave peer to peer
Foundation Fieldbus HSE	estrella	par trenzado fibra óptica	240 p/segm 32.768 sist	100M	0.1 par 2 fibra	Single/multi master
Foundation Fieldbus H1	estrella o bus	par trenzado fibra óptica	240 p/segm 32.768 sist	31.25K	1.9 cable	Single/multi master
LonWorks	bus, anillo, lazo, estrella	par trenzado fibra óptica coaxial, radio	32768/dom	500K	2	Master/Slave peer to peer
Interbus-S	segmentado	par trenzado fibra óptica	256 nodos	500K	400/segm 12.8 total	Master/Slave
DeviceNet	troncal/puntual c/bifurcación	par trenzado fibra óptica	2048 nodos	500K	0.5 6 c/repetid	Master/Slave, multi-master, peer to peer
AS-I	bus, anillo, arbol, estrella	par trenzado	31 p/red	167K	0.1, 0.3 c/rep	Master/Slave
Modbus RTU	linea, estrella, arbol, red con segmentos	par trenzado coaxial radio	250 p/segm	1.2 a 115.2K	0.35	Master/Slave
Ethernet Industrial	bus, estrella, malla-cadena	coaxial par trenzado fibra óptica	400 p/segm	10, 100M	0.1 100 mono c/switch	Master/Slave peer to peer
HART		par trenzado	15 p/segm	1.2K		Master/Slave

Master/Slave: Maestro/Esclavo
Peer to Peer: Punto a Punto
Multi-Master: Multi Maestro

Figura 18. Protocolos y buses de campos industriales y sus características.

2.3.1 DeviceNet

Bus basado en CAN. Su capa física y capa de enlace se basan en ISO 11898 y en la especificación de Bosh 2.0. DeviceNet define una de las más sofisticadas capas de aplicaciones industriales sobre bus CAN. Fue desarrollado por Allen-Bradley a mediados de los noventa, aunque posteriormente pasó a ser una especificación abierta soportada en la ODVA (Open DeviceNet Vendor Association). Cualquier fabricante puede asociarse a esta organización y obtener especificaciones, homologar productos, etc. Es posible la conexión de hasta 64 nodos con velocidades de 125 Kbps a 500 Kbps en distancias de 100 a 500 m.

Utiliza una definición basada en orientación a objetos para modelar los servicios de comunicación y el comportamiento externo de los nodos. Define mensajes y conexiones para funcionamiento maestro-esclavo, interrogación cíclica o lanzamiento de interrogación general de dispositivos, mensajes espontáneos de cambio de estado, comunicación uno-uno, modelo productor-consumidor, carga y descarga de bloques de datos y ficheros, etc.

DeviceNet ha conseguido una significativa cuota de mercado. Existen más de 300 productos homologados y se indica que el número de nodos instalados supera los 300.000 en 1998. Está soportado por numerosos fabricantes: Allen-Bradley, ABB, Danfoss, Crouzet, Bosh, Control Techniques, Festo, Omron, entre otros.

2.3.2 Lonworks

La empresa Echelon, localizada en California, fue fundada en 1988. Comercializa el bus de campo LonWorks basado en el protocolo LonTalk y soportado sobre el NeuronChip. Alrededor de estas marcas ha construido toda una estructura de productos y servicios, hábilmente comercializados, dirigidos al mercado del control distribuido en domótica, edificios inteligentes y control industrial.

El protocolo LonTalkcubre cubre todas las capas OSI. Se soporta en hardware y firrmware sobre el NeuronChip. Se trata de un microcontrolador que incluye el controlador de comunicaciones y toda una capa de firmware que, además de implementar el protocolo, ofrece una serie de servicios que permiten el desarrollo de aplicaciones en el lenguaje Neuron C, una variante de ANSI C. Motorola y Toshiba son los fabricantes de NeuronChip.

Además, Echelon brinda la posibilidad de abrir la implementación de LonWorks a otros procesadores. La red Lonworks ofrece una variada selección de medios físicos y topologías de red: par trenzado en bus, anillo y topología libre, fibra óptica, radio, transmisión sobre red eléctrica, etc. El soporte más usual es par trenzado a 38 o 78 Kbps. Se oferta una amplia gama de servicios de red que permiten la construcción de extensas arquitecturas con multitud de nodos, dominios y grupos, típicas de grandes edificios inteligentes.

El método de comparación de medio es acceso CSMA predictivo e incluye servicios de prioridad de mensajes. Echelon ofrece herramientas de desarrollo, formación, documentación y soporte técnico. Echelon basa su negocio en la comercialización del bus, medios, herramientas y soporte.

2.3.2 Foundation Fieldbus

Un bus orientado sobre todo a la interconexión de dispositivos en industrias de proceso continuo. Su desarrollo ha sido apoyado por importantes fabricantes de instrumentación (Fisher-Rosemount, Foxboro, entre otros). En la actualidad existe una asociación de fabricantes que utilizan este bus que gestiona el esfuerzo normalizador.

Normalizado como ISA SP50, IEC-ISO 61158 –ISA es la asociación internacional de fabricantes de dispositivos de instrumentación de proceso–. En su nivel H1 (uno) de la capa física sigue la norma IEC 11158-2 para comunicación a 31,25 Kbps. Es, por tanto, compatible con Profibús PA, su principal competidor. Presta especial atención a las versiones que cumplen normas de seguridad intrínseca para industrias de proceso en ambientes combustibles o explosivos. Se soporta sobre par trenzado y es posible la reutilización de los antiguos cableados de instrumentación analógica 4-20 mA.

Se utiliza comunicación síncrona con codificación Manchester Bifase-L. La capa de aplicación emplea un protocolo sofisticado, orientado a objetos con múltiples formatos de mensaje. Distingue entre dispositivos con capacidad de arbitración (Link Master) y normales. En cada momento un solo Link master arbitra el bus, puede ser sustituido por otro en caso de fallo. Utiliza diversos mensajes para gestionar la comunicación por paso de testigo, comunicación cliente-servidor, modelo productor-consumidor, etc.

Existen servicios para configuración, gestión de diccionario de objetos en nodos, acceso a variables, eventos, carga y descarga de ficheros y aplicaciones, ejecución de aplicaciones, etc. La codificación de mensajes se define según ASN.1. El nivel H2 (dos) está basado en Ethernet de alta velocidad (100 Mbps) y orientado al nivel de control de la red industrial.

2.4 El Conflicto de los Buses

"Ante la variedad de opciones existentes, parece razonable pensar que fabricantes y usuarios hicieran un esfuerzo en la búsqueda de normativas comunes para la interconexión de sistemas industriales. Lo que ha venido llamándose 'la guerra de los buses' tiene que ver con la permanente confusión reinante en los entornos normalizadores en los que se debate la especificación del supuesto 'bus de campo universal'.

Desde mediados de los años 80, la Comisión Electrotécnica Internacional (IECCEI) y la Sociedad de Instrumentación Americana (ISA) ha sido escenario del supuesto esfuerzo de los fabricantes para lograr el establecimiento de una norma única de bus de campo de uso general. En 1992 surgieron dos grupos, el ISP (Interoperable SystemsProject) y WorldFIP cada uno promoviendo su propia versión del bus de campo. En el primer grupo estaban fabricantes como Siemens, Fisher- Rosemount, Foxboro y Yokogawa. En el segundo Allen-Bradley, HoneyWell, Square D y diversas empresas francesa. En 1994 ambos grupos se unieron en la Fieldbus Foundation. El debate se trasladó luego, y continúa en la actualidad, a la conjunción de Fieldbus y el mundoProfibus. Los años pasan y la norma del supuesto bus universal nunca se acaba de generar y en el camino aparecen nuevas opciones como CAN, LonWorks y Ethernet. Incluso el debate es confuso y totalmente incomprensible, pues otras empresas participantes en el debate generaban en paralelo soluciones propias. Es el caso de Allen-Bradley con DeviceNet y HoneyWell con SDS. La realidad es que sólo los usuarios están realmente interesados en la obtención de normas de uso general. Los fabricantes luchan por su cuota de mercado y, en general, sólo están a favor de una norma cuando ésta recoge las características de su propia opción, lo cual es comprensible dadas las fuertes inversiones necesarias para el desarrollo de un bus industrial normalizado. El debate sigue abierto". [7]

[7] Kaschel y Pinto, p. 8

Capítulo 3. Desarrollo de una central de medida con comunicaciones Modbus TCP y de una aplicación para su supervisión

3 Introducción

En el presente capítulo se exponen los pasos que se han de seguir para implementar una central de medida de parámetros eléctricos con comunicación Modbus TCP y crear una aplicación de escritorio que recoja los datos de dicha central para su posterior manipulación por el usuario final.

El sistema consta de un pequeño contador de Potencia consumida con una salida de pulsos, un circuito basado en un microcontrolador que cuenta dichos pulsos y que implementa un servidor Modbus TCP, y una aplicación realizada en Java que recoge los datos del servidor y los muestra en un formato legible para el usuario final.

3.1 Elección del Hardware

El hardware utilizado para la implementación del contador de pulsos con Modbus TCP es la Wifi Demo G board de Microchip. Estas son las características en base a las cuales se he elegido el dispositivo:

- Microcontrolador **PIC32MX695F512H**: Este microntrolador de 32 bits cuenta con la posibilidad de establecer comunicaciones vía Ethernet y Wifi, conversores AD, memoria suficiente para albergar el programa y bajo consumo. Estas son sus características principales:
    - 80MHz/105DMIPS
    - 10/100 Ethernet MAC
    - 512K Flash
    - 128K RAM
    - Conversor AD 10-bit.

Figura 19. PIC 32MX695F512H

- Módulo Wifi **MRF24WG0MA**: Certificado según IEEE 802.11 b/g , antena integrada en el PCB. Es una solución ideal para aplicaciones de bajo consumo y con una tasa de transferencia de datos también baja, como redes de sensores para automatización industrial o domótica.
    - IEEE 802.11 b/g Wi-Fi
    - Wi-Fi Direct
    - Data Rate: 1, 2, 5.5, 11 Mbps 802.11b; 6, 9, 12,18, 24, 36, 48, 54 Mbps 802.11g
    - Compatible con redes IEEE 802.11b/g/n
    - Tamaño pequeño: 21 mm x 31 mm
    - Soporta modo bajo consumo
    - Antena PCB integrada

Figura 20. Módulo Wifi MRF24WG0MA.

- Seguridad WEP, WPA-PSK y WPA2-PS

- **Entradas y salidas digitales:** La Wifi g Demo Board tiene algunas entradas/salidas cableadas al microcontrolador a las que se puede conectar cualquier tipo de sensor, ya sea una sonda con salida analógica, con salida digital o cualquier dispositivo que entregue una señal digital –como finales de carrera o sensores de presencia–.

Figura 21. Wifi G demo bord de Microchip.

3.2 Elección del Software

3.2.1 Elección del Software Necesario para la Programación del Microcontrolador

Como se expone en el Capítulo 2, existen multitud de librerías software para implementar el protocolo Modbus. A continuación se muestran las usadas en este proyecto para cada extremo de la comunicación.

Puesto que se ha elegido la Versión TCP de Modbus es imprescindible contar con una implementación de la pila de protocolos TCP/IP. En el caso del microntrolador se ha empleado el Stack TCP/IP de Microchip, compañía que ofrece este conjunto de librerías con multitud de proyectos y ejemplos a partir de los cuales se puede comenzar a trabajar. Dada la complejidad de este software no es conveniente utilizarlo como librerías que se incluyen en el proyecto, sino como un proyecto completo que podemos adaptar a nuestras necesidades.

A continuación se muestra la estructura de uno de estos proyectos ejemplo, detallando los ficheros en los que se encuentran los parámetros a configurar y aquellos en los que podemos escribir nuestro código.

3.2.2 Elección del Software Necesario para la Programación de la Aplicación

Para la aplicación en la que se muestran los datos al usuario final se ha empleado el lenguaje de programación Java. En este caso no existe ningún problema para implementar TCP/IP pues, al ser un lenguaje de alto nivel, cuenta con multitud de librerías que implementan esta pila de protocolos y que forman, a su vez, parte del propio lenguaje.

En la implementación de ModbusTCP se ha elegido JAMOD, una librería muy completa y estable que implementa ambas versiones del protocolo (TCP, RTU). Esta librería cuenta, además, con una completa documentación.

3.2.3 Software Adicional Empleado en el Proyecto

Para el desarrollo del presente proyecto han sido de especial utilidad algunas aplicaciones, tanto en la fase de desarrollo como en la de detección y corrección de errores.

MPLAX IDE: Es el entorno de desarrollo de Microchip para sus microcontroladores. Está basado en el conocido IDE de Java Netbeans. Permite escribir código en C y C++, compilarlo y gravarlo en el microcontrolador.

Microchip aplication libraries: Es el conjunto de librerías para desarrollo de aplicaciones de Microchip. Contiene, además del Stack TCP, código para operar sobre todo tipo de periféricos como sensores, conversores AD/DA; y para comunicar utilizando distintos medios físicos como USB, Ethernet, SPI, UART.

TCP/IP Configuration Wizard: Es una pequeña pero muy útil aplicación que ayuda al programador a configurar el Stack TCP de Microchip a través de un entorno gráfico y de una forma mucho más intuitiva que manipulando el código fuente.

XC32: Es el nuevo compilador de microchip para microcontroladores de 32 bits. Permite compilar código en C y en C++. Si bien es posible utilizarlo por línea de comandos, es más recomendable instalarlo dentro del IDE MPLABX, pues facilita de manera significativa su uso y su configuración.

ECLIPSE IDE: Es uno de los IDE's ms conocidos para desarrollo de aplicaciones en Java. Permite ordenar el proyecto de forma higiénica, incluir librerías rápida y eficazmente y depurar las aplicaciones de forma intensiva. Además, Eclipse cuenta con la posibilidad de instalar multitud de complementos que facilitan tareas arduas como la confección de interfaces gráficas.

CAS MODBUS CLIENT: Es una aplicación creada por la empresa Chipkin, dedicada al desarrollo de dispositivos de automatización industrial. Puede descargase gratuitamente en su web www.chipkin.com. Implementa un cliente Modbus, permite crear conexiones Modbus RTU y TCP, lanzar consultas sobre ellas y visualizar la respuesta en caso de que la comunicación se haya producido sin errores. Esta aplicación es muy útil durante la fase de desarrollo de proyectos que impliquen la utilización de Modbus porque ayuda a depurar los posibles errores de programación.

Wireshark: Es el analizador de protocolos con un uso más extendido. Wireshark es una herramienta muy útil para el desarrollo de aplicaciones con protocolos de comunicación. En el caso particular de este proyecto ha sido muy adecuado puesto que es capaz de reconocer el protocolo Modbus y analizar sus tramas en profundidad.

3.3 Programación y Depuración del Microcontrolador

Como se ha expuesto en el apartado anterior, el IDE de desarrollo empleado para la programación del microcontrolador es el MPLABX. A continuación se muestra el proceso que se ha seguido para crear el proyecto con el Stack TCP de Microchip, incluir las comunicaciones Modbus TCP y programar el comportamiento deseado para nuestro dispositivo.

3.3.1 Creación del Proyecto en MPLABX

Lo principal a la hora de realizar un proyecto que proporcione conectividad TCP IP, ya sea a través de Ethernet o de Wifi, es configurar el Stack TCP. Dada la complejidad de este firmware, Microchip ofrece multitud de proyectos ejemplo a partir de los cuales se puede empezar a trabajar para modificarlos a nuestro gusto y según nuestras necesidades.

En este caso el proyecto que mejor se adapta es WIFI_G_Demo_Board que podemos encontrar dentro de la carpeta microchip_solutions_v2013-06-15 en la cual se instalan todas las librerías para desarrollo de aplicaciones de microchip. Este proyecto está configurado para el módulo Wifi MRF24WG0MA.

En la siguiente figura se puede observar la estructura del proyecto. El directorio Header Files contiene los fichero cabecera en los que se definen funciones, tipos de datos, etc., mientras que el directorio Source Files contiene todo el código fuente que implementa la funcionalidad del Stack.

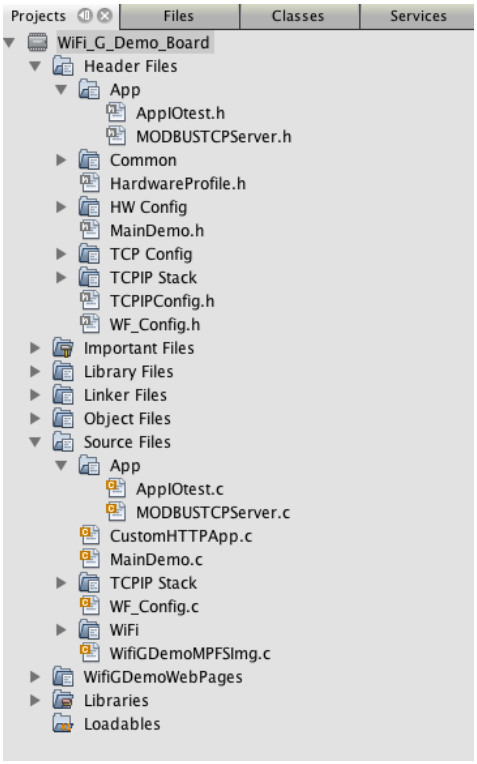

Figura 22. Estructura del proyecto en MPLABX.

Estos son los ficheros más importantes a la hora de configurar el Stack:

TCPIP MRF24W:

Este fichero se encuentra en el directorio TCP Config. Contiene información sobre los elementos y funcionalidades del Stack que el usuario desea compilar. Las siguientes líneas de código activan o desactivan los protocolos y servicios que el usuario puede utilizar, como ICMP, DHCP o MODBUS_TCP.

```
2. #define STACK_USE_MODBUS_TCP_SERVER
3. //#define STACK_USE_UART //
4. //#define STACK_USE_UART2TCP_BRIDGE //
6. #define STACK_USE_ICMP_SERVER //
7. //#define STACK_USE_ICMP_CLIENT //
8. #define STACK_USE_HTTP2_SERVER //
9. //#define STACK_USE_SSL_SERVER //
10. //#define STACK_USE_SSL_CLIENT //
11. //#define STACK_USE_AUTO_IP //
12. #define STACK_USE_DHCP_CLIENT //
13. #define STACK_USE_DHCP_SERVER //
14. //#define STACK_USE_FTP_SERVER //
15. //#define STACK_USE_SMTP_CLIENT //
16. //#define STACK_USE_SNMP_SERVER //
17. //#define STACK_USE_SNMPV3_SERVER //
18. //#define STACK_USE_TFTP_CLIENT //
19. //#define STACK_USE_TELNET_SERVER //
20. #define STACK_USE_ANNOUNCE //
21. #define STACK_USE_DNS //
22. #define STACK_USE_DNS_SERVER //
23. #define STACK_USE_NBNS //
24. #define STACK_USE_REBOOT_SERVER //.
25. //#define STACK_USE_SNTP_CLIENT //
26. //#define STACK_USE_UDP_PERFORMANCE_TEST //
27. //#define STACK_USE_TCP_PERFORMANCE_TEST //
28. //#define STACK_USE_DYNAMICDNS_CLIENT //
29. //#define STACK_USE_BERKELEY_API //
30. #define STACK_USE_ZEROCONF_LINK_LOCAL //
31. #define STACK_USE_ZEROCONF_MDNS_SD //
```

En este fichero también se definen parámetros como el nombre de host, dirección IP, mascara de subred, puerta de enlace, además de la configuración referente a la distribución de memoria para almacenamiento de páginas web o ficheros.

```
1. #define  MY_DEFAULT_HOST_NAME "MCHPBOARD"
2.
3. #define  MY_DEFAULT_MAC_BYTE1 (0x00) //
4. #define  MY_DEFAULT_MAC_BYTE2 (0x04) //
5. #define  MY_DEFAULT_MAC_BYTE3 (0xA3) //
6. #define  MY_DEFAULT_MAC_BYTE4 (0x00) //
7. #define  MY_DEFAULT_MAC_BYTE5 (0x00) //
8. #define  MY_DEFAULT_MAC_BYTE6 (0x00) //
9.
10. #define  MY_DEFAULT_IP_ADDR_BYTE1 (192ul)
11. #define  MY_DEFAULT_IP_ADDR_BYTE2 (128ul)
12. #define  MY_DEFAULT_IP_ADDR_BYTE3 (1ul)
13. #define  MY_DEFAULT_IP_ADDR_BYTE4 (3ul)
14.
15. #define  MY_DEFAULT_GATE_BYTE1 (192ul)
16. #define  MY_DEFAULT_GATE_BYTE2 (128ul)
17. #define  MY_DEFAULT_GATE_BYTE3 (1ul)
18. #define  MY_DEFAULT_GATE_BYTE4 (3ul)
19.
20. #define  MY_DEFAULT_PRIMARY_DNS_BYTE1 (192ul)
21. #define  MY_DEFAULT_PRIMARY_DNS_BYTE2 (128ul)
22. #define  MY_DEFAULT_PRIMARY_DNS_BYTE3 (1ul)
23. #define  MY_DEFAULT_PRIMARY_DNS_BYTE4 (3ul)
24.
25. #define  MY_DEFAULT_MASK_BYTE1 (255ul)
26. #define  MY_DEFAULT_MASK_BYTE2 (255ul)
27. #define  MY_DEFAULT_MASK_BYTE3 (0ul)
28. #define  MY_DEFAULT_MASK_BYTE4 (0ul)
29.
30.
31. #define  MY_DEFAULT_SECONDARY_DNS_BYTE1 (0ul)
32. #define  MY_DEFAULT_SECONDARY_DNS_BYTE2 (0ul)
```

```
#define  MY_DEFAULT_SECONDARY_DNS_BYTE3 (0ul)
#define  MY_DEFAULT_SECONDARY_DNS_BYTE4 (0ul)
```

Existen otros muchos parámetros que se pueden configurar a través de este fichero pero, dada su complejidad, es más recomendable el uso de la aplicación TCP/IP Configuration Wizard de Microchip que hace mucho más sencillo este proceso. Esta aplicación nos permite configurar el fichero TCPIP MRF24W cómodamente a través de una interfaz gráfica intuitiva.

HWP MRF24W XC32:

Es el fichero en el que se configura el conexionado del microcontrolador a los elementos periféricos como leds, pulsadores y el módulo Wifi MRF24WG0MA.

El siguiente código nombra las patillas del microcontrolador que son usadas para los leds y el pulsador que incorpora el prototipo. También configura la entrada analógica utilizada para sondear el estado de la batería.

```
//----------------------------
// LED and Button I/O pins
//----------------------------
#define LED0_TRIS (TRISEbits.TRISE0) // Ref D10 Green
#define LED0_IO (LATEbits.LATE0)
#define LED1_TRIS (TRISFbits.TRISF1) // Ref D9 Yellow
#define LED1_IO (LATFbits.LATF1)
#define LED2_TRIS (TRISFbits.TRISF0) // Ref D8 Red
#define LED2_IO (LATFbits.LATF0)

#define LEDS_ON() {LED0_ON(); LED1_ON(); LED2_ON();}
#define LEDS_OFF() {LED0_OFF(); LED1_OFF(); LED2_OFF();}

#define LED0_ON() LATESET = BIT_0;
#define LED0_OFF() LATECLR = BIT_0;
#define LED0_INV() LATEINV = BIT_0;

#define LED1_ON() LATFSET = BIT_1;
#define LED1_OFF() LATFCLR = BIT_1;
#define LED1_INV() LATFINV = BIT_1;

#define LED2_ON() LATFSET = BIT_0;
#define LED2_OFF() LATFCLR = BIT_0;
#define LED2_INV() LATFINV = BIT_0;

#define SW0_TRIS (TRISDbits.TRISD9)
#define SW0_IO (PORTDbits.RD9)

#define VBAT_TRIS (TRISBbits.TRISB0) // Battery level ADC input
```

En esta otra zona del código se define el nombre que se da a las patillas conectadas al módulo MRF24WG0MA. La comunicación entre el microcontrolador y el modulo Wifi se lleva a cabo mediante el protocolo serie SPI, el cual requiere tres conexione para envío y recepción de datos y una señal de reloj

```
//----------------------------
// MRF24WG0MA/B WiFi I/O pins
//----------------------------

#define WF_CS_TRIS (TRISGbits.TRISG9)
#define WF_CS_IO (LATGbits.LATG9)
```

```
#define WF_SDI_TRIS (TRISGbits.TRISG7)
#define WF_SCK_TRIS (TRISGbits.TRISG6)
#define WF_SDO_TRIS (TRISGbits.TRISG8)
#define WF_RESET_TRIS (TRISDbits.TRISD1)
#define WF_RESET_IO (LATDbits.LATD1)
```

MainDemo.c:

Este fichero representa el punto de entrada de a la aplicación, puesto que es el que contiene la función main(). En esta función se configuran los periféricos necesarios para el correcto funcionamiento del prototipo. Tras esta configuración inicial el programa entra en un bucle infinito en el que se van atendiendo, una a una, todas las peticiones entrantes a modo de máquina de estados. Este funcionamiento asegura que la ejecución del programa no se bloquee en ningún momento. Debido a su tamaño este fichero está incluido en el *Anexo 1*.

"Una vez que todos los ítems han sido inicializados comienza un bucle multitarea cooperativo. Este bucle infinito ejecuta todas las tareas relacionadas con el Stack. Las funciones creadas por el usuario deben ser añadidas al final de este bucle. Nótese que se emplea un sistema multitarea cooperativo, con lo que las tareas programadas por el usuario no deben ser demasiado largas o entorpecerán el correcto funcionamiento de programa".[8]

MODBUSTCPServer.c:

Implementa el servidor ModbusTCP en la aplicación. El fichero se encuentra en el *Anexo 1*, pero a continuación se muestra el fichero de cabecera MODBUSTCPServer.h como resumen de las funciones y tipos de datos definidos para dicho servidor.

```
#define MODBUS_PORT 502

#define INPUT_REG_SIZE 25
#define MODBUS_RX_BUFFER_SIZE 260
#define MODBUS_TX_BUFFER_SIZE 50
#define COIL_SIZE 5

#define ReadCoil 1
#define ReadDiscreteInputs 2
#define ReadHoldingRegister 3
#define ReadInputRegister 4
#define WriteSingleCoil 5
#define WriteSingleRegister 6
#define WriteMultipleCoils 15
#define WriteMultipleRegister 16

#define Illegal_Function_Code 0x01u
#define Illegal_Data_Address 0x02u

#define MODBUS_UnitID 6
#define MODBUS_FunctionCode 7
#define MODBUS_BYTE_COUNT 8
#define MODBUS_DataStart 13
#define MODBUS_ExceptionCode 8

typedef struct{
WORD_VAL TransactionID; //Transaction ID
WORD_VAL ProtocolID; //Protocol ID
WORD Length; //Length
BYTE UnitID; //Unit ID
BYTE FunctionCode; //Function code
```

8 Fichero main.c

```
43. WORD_VAL StartAddress; //Starting register address
44. WORD_VAL NumberOfRegister; //Number of registers
45. } _MODBUS_COMMAND; //MODBUS_COMMAND
47.
48. typedef struct{
49. BYTE Val;
50. BYTE Addr;
51. } WORD_VAL1, WORD_BITS1;
53.
55. void MODBUSTCPServer(void);
56. void ProcessReceivedMessage(void);
57. void readHoldingRegister(void);
58. void writeHoldingRegister(void);
59. void readInputRegister(void);
60. void writeMultipleCoils(void);
61. void writeSingleCoil(void);
62. void ModbusError(unsigned char ErrNum);
63.
64. #endif
```

- **MODBUS_PORT**: Es el puerto TCP que va a utilizar el servidor para escuchar las peticiones entrantes.

- **INPUT_REG_SIZE 25, MODBUS_RX_BUFFER_SIZE, MODBUS_TX_BUFFER_SIZE, COIL_SIZE :** Define los tamaños de las áreas de memoria de entradas (registros y bits) y los tamaños de los buffers de entrada y salida.

- Las **líneas 13-20** definen las funciones Modbus que se van a implementar para este servidor.

- **Illegal_Function_Code, Illegal_Data_Address:** Definen los códigos de función que se han implementado en el proyecto.

- Las **líneas de la 29 a al 33** definen las posiciones que ocupan los campos más importantes de las consultas Modbus que recibe el servidor, como el código de función, el identificador de dispositivo o el comienzo de los datos.

- **_MODBUS_COMMAND:** Estructura que contiene todos los datos necesarios para conformar una trama Modbus TCP.

- **MODBUSTCPServer():** Es la función que implementa el servidor ModbusTCP. Funciona como una máquina de estados para no bloquear al resto de servicios del Stack. Esta función es llamada desde el bucle infinito que se encuentra en la función main() para ir sondeando de manera cíclica los mensajes o eventos relacionados con el servidor Modbus TCP.

- **ProcessReceivedMessage:** Procesa los mensajes Modbus entrantes llamando a la función que corresponda, dependiendo del tipo de dato solicitado.

El resto de funciones que aparecen a continuación son las que resuelven las peticiones Modbus de cada una de las funciones soportadas por el servidor.

AppIOtest:

Es el fichero en el que se define la lógica de la aplicación que se ejecuta en el dispositivo. En la primera línea, mediante el modificador extern, se obtiene una referencia al array HOLGING_REG definido en **MainDemo.c**, que contiene los registros de 16 bits que maneja Modbus para este dispositivo.

```
1.  extern WORD HOLDING_REG[20];
```

La función **IO-Pull_Task()** es llamada periódicamente en el bucle infinito que se encuentra en la función main(). También está implementada como una máquina de estados con el fin de no bloquear el resto de la aplicación. La lógica de la aplicación es muy sencilla pero extrapolable a todo tipo de soluciones prácticas:

- El registro HOLDING_REG[3] se incrementa en una unidad cada vez que el pulsador conectado al microcontrolador es presionado.

- Los leds conectados al microcontrolador parpadean con una frecuencia igual a ½ del valor que contiene el registro.

```
7. void IO_Pull_Task(void){
8.
9. static enum _sm_pulsador{
10. wait_low = 0,
11. wait_hi
12. }sm_pulsador=wait_low;
13.
14. ((LED0_IO)?(HOLDING_REG[0]=1):(HOLDING_REG[0]=0));
15. ((LED1_IO)?(HOLDING_REG[1]=1):(HOLDING_REG[1]=0));
16. ((LED2_IO)?(HOLDING_REG[2]=1):(HOLDING_REG[2]=0));
17.
18. switch(sm_pulsador){
19.             case wait_low:
20.                     if(SW0_IO)break;
21.                     sm_pulsador=wait_hi;
22.             break;
23.             case wait_hi:
24.                     if(!SW0_IO)break;
25.                     HOLDING_REG[3]++;
26.                     sm_pulsador=wait_low;
27.              break;
28. }
32.
33. ((SW0_IO)?(HOLDING_REG[6]=1):(HOLDING_REG[6]=0));
34.
35.
36.
37. if((TickGet() - BlinkTimeLed1) >= TICK_SECOND * (DWORD)HOLDING_REG[7])
{
38.              LED1_IO = !LED1_IO;
39.             BlinkTimeLed1 = TickGet();
40. }
41.
42. if((TickGet() - BlinkTimeLed2) >= TICK_SECOND * (DWORD)HOLDING_REG[8])
{
43.             LED2_IO = !LED2_IO;
44.             BlinkTimeLed2 = TickGet();
45. }
46.
47. }
```

La función **IO_Pull_Init()** pone a cero todas las posiciones de memoria de HOLDING_REG.

```
50. void IO_Pull_Init(void){
51.
52. HOLDING_REG[0] = 0;
53. HOLDING_REG[1] = 0;
54. HOLDING_REG[2] = 0;
```

```
55. HOLDING_REG[3] = 0;
56. HOLDING_REG[4] = 0;
57. HOLDING_REG[5] = 0;
58. HOLDING_REG[6] = 0;
medioperromn
```

3.3.2 Depuración y Pruebas sobre la Aplicación en el Microcontrolador

Para comprobar el correcto funcionamiento de la aplicación es muy útil la utilización de un software cliente Modbus como CAS. Con esta aplicación podemos enviar peticiones al servidor tanto de escritura como de escritura.

En la siguiente figura se muestra una captura de CAS Modbus en la que se observa una operación de lectura sobre el registro de retención número 3.

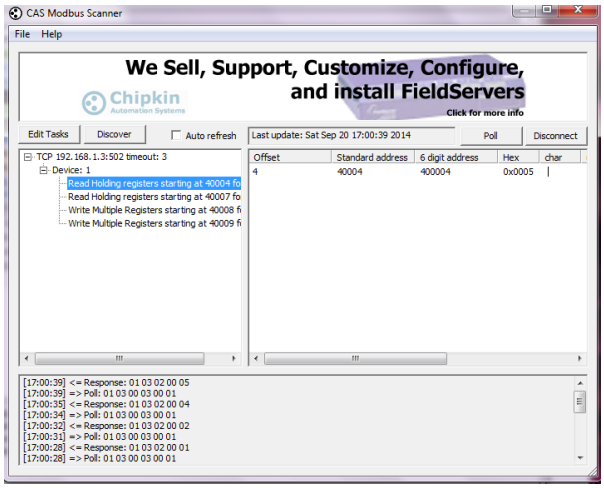

Figura 22. Aliente Modbus CAS de Chipkin.

En la columna de la izquierda aparecen las peticiones generadas por el usuario, en la de la derecha se observa el resultado en caso de que la comunicación haya sido satisfactoria. La zona inferior de la aplición es de especial utilidad porque en ella se pueden ver los mensajes Modbus intercambiados.

Si durante la depuración de la aplicación hay problemas relacionados con las capas de transporte o red de la pila TPC/IP resulta muy útil emplear Wireshark, ya que es capaz de analizar todos los protocolos de comunicación que intervienen en la transacción.

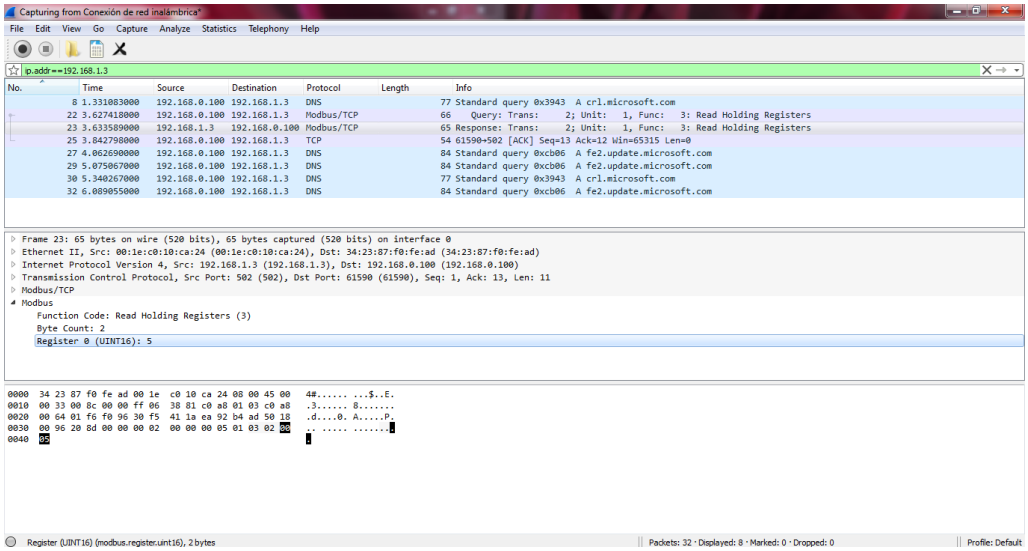

Figura 23. Transacciones Modbus capturadas en Wireshark.

3.4 Programación y Depuración de la Aplicación Cliente.

Esta aplicación implementa un cliente ModBusTCP para la lectura del dispositivo descrito en el apartado anterior. Para su creación se ha escogido Eclipse como entorno de desarrollo.

El código completo de las clases más relevantes de la aplicación está incluido en el *Anexo 2*. A continuación se expone detalladamente el proceso seguido para la programación de la parte referente a la comunicación ModbusTCP.

3.4.1 Creación del Proyecto en Eclipse

Para la comunicación Modbus TCP de la aplicación cliente se ha utilizado la librería Jamod. Este software puede descargarse directamente de la web jamod.sourceforge.net. Una vez incluido el fichero jamod-1-2-SNAPSHOT.jar se tiene acceso a todas las clases de la librería.

La aplicación consta de una tabla en la que se puede visualizar el contenido de registros del dispositivo, leídos mediante Modbus TCP, y una zona de configuración en la que se pueden dar de alta conexiones con dispositivos y registros a leer sobre dichas conexiones.

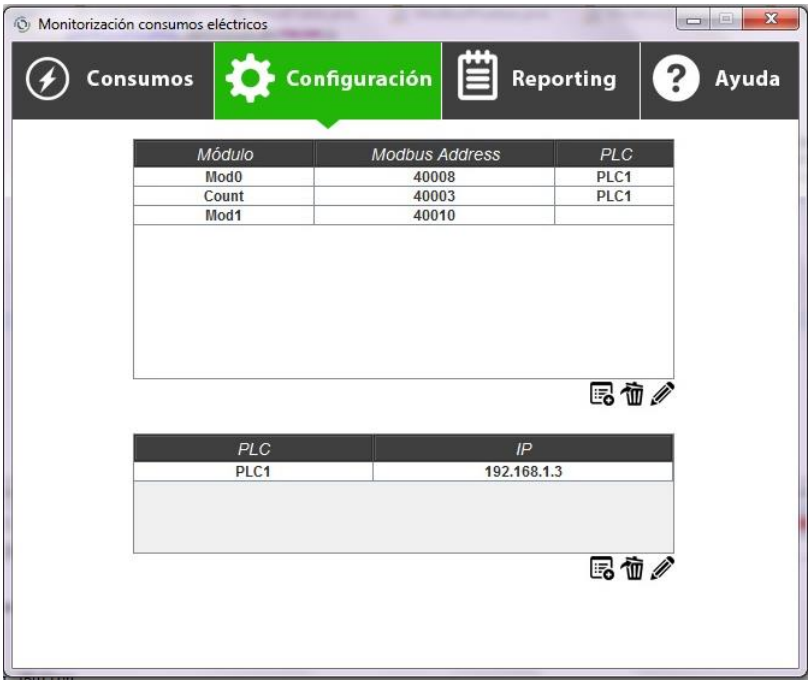

Figura 24. Pantalla de configuración de la aplicación cliente Modbus TCP.

La interfaz de usuario se ha programado mediante Swing, el conjunto de librerías de Java destinado a tal efecto.

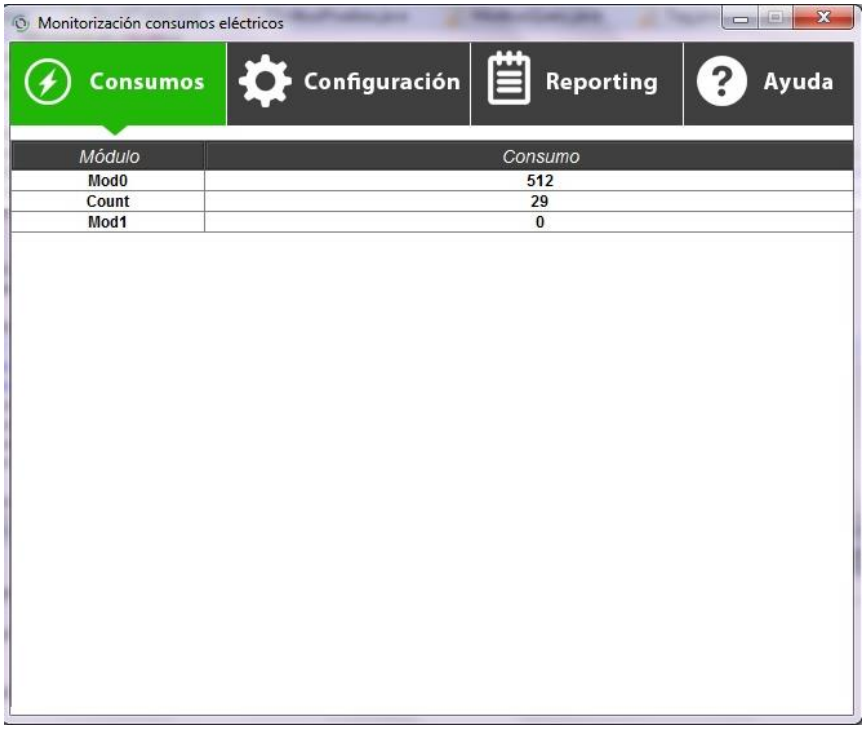

Figura 25. Visualización de valores de registros leídos a través de Modbus en la aplicación cliente.

3.4.2 Implementación del Cliente Modbus

Para la implementación del cliente Modbus TCP de la aplicación se han seguido los siguientes pasos:

- Creación de una clase Java que configura la conexión, crea la petición y la ejecuta para obtener el valor recibido del servidor y mostrarlo en la interfaz gráfica.

- Creación de una clase java que implementa un hilo de ejecución distinto del principal para que la comunicación Modbus con el dispositivo no interfiera con la ejecución del resto de la aplicación.

- Creación de una clase que se encarga de guardar los datos persistentes de la aplicación como la configuración de la conexión y de los registros a leer.

3.4.2.1 La clase ModbusQuery.

Esta clase ha sido programada siguiendo el patrón de diseño singleton. Este patrón asegura que solo se pueda crear una instancia del objeto que implementa la clase. De esta manera se asegura que la conexión TCP pueda ser accedida desde cualquier zona de la aplicación.

```
private synchronized static void createInstance(String addr,int port,int ref, int count,int repeat) throws Exception {
179. if (INSTANCE == null) {
180. INSTANCE = new ModbusQuery(addr,port,ref,count,repeat);
181. }
182. }
183.
184. public static ModbusQuery getInstance(String addr,int port,int ref, int count,int repeat) throws Exception {
185. if (INSTANCE == null) createInstance(addr,port,ref,count,repeat);
186. return INSTANCE;
187. }
```

Una vez que se obtiene una instancia de la clase mediante el método getInstance(), queda abierta la conexión con el servidor. A partir de este momento se pueden enviar peticiones de lectura hacia el servidor mediante el método ExecuteSingleQuery().

```
155. public int ExecuteSingleQuery() throws Exception{
154. int k = 0;
156. int result;
157.
158. con.connect();
159.
160. //Prepare the request
161. req = new ReadMultipleRegistersRequest(this.ref,1);
162. req.setUnitID(1);
163. //Prepare the transaction
164. trans = new ModbusTCPTransaction(con);
165. trans.setRequest(req);
166.
167.
168. trans.execute();
169.
170. res = (ReadMultipleRegistersResponse) trans.getResponse();
171.
172. result=(int)res.getRegisterValue(0);
```

```
173.
174. return result;
175.
176. }
177.
```

Este método envía hacia el servidor una petición ModbusTCP de lectura de múltiples registros. Los valores de dirección de memoria y número de registros a leer son configurables en la creación del objeto o mediante métodos get() y set(). El valor de retorno de ExecuteSingleQuery() es el recibido como respuesta del servidor.

3.4.2.2 Programación de la clase Consumos.

Es la clase encargada de representar los datos leídos del servidor. Hereda de la clase Java JPanel y contiene una tabla en la cual se muestran los resultados de la petición. Mediante la implementación de la interface Runnable se inicia un nuevo hilo de ejecución en el cual se recogen los datos recibidos del servidor para actualizar la tabla.

Mediante los métodos start() y stop() se puede iniciar y detener la ejecución del hilo en el que se realizan las peticiones al servidor. En el caso de que se detenga la ejecución del hilo, se cierra la conexión con el método close() de la clase ModbudQuery para evitar que queden conexiones abiertas al abandonar la aplicación.

```
49. @Override
50. public void run() {
51.         int i=0;
52.          while(!stop){
53.                         i++;
54.
55.                         System.out.println("Iteracion:"+i);
56.                         tableModel=null;
57.                         table=null;
58.                          this.remove(scrollPanel);
59.                          try {
60.                                         tableModel=new TableModel(ResultTable.getRegisters(),columnNames);
61.                                         table =new ResultTable(tableModel);
62.                                         scrollPanel=new JScrollPane(table);
63.                                         scrollPanel.setBounds(0, 0, 654, 455);
64.                                         table.setFillsViewportHeight(true);
65.                                         setLayout(null);
66.                                         this.add(scrollPanel);
67.                                         this.setVisible(true);
68.
69.
70.                                         Thread.sleep(100);
71.
72.                                         }catch (Exception e) {e.printStackTrace(); }
73.
74.                                         }
75.                                         try {
76.                                                         ModbusQuery.getInstance(null, 0, 0, 0, 0).close();
77.                                         } catch (Exception e) {
78.
79.                                         e.printStackTrace();
80.                                         }
81.
82.         }
83. }
```

3.4.2.3 Programación de la clase Config.

Es la clase que se encarga de mantener los datos persistentes de la aplicación en una base de datos SQlite para que no haya que configurar los datos de las conexiones y registros de lectura cada vez que se inicia la aplicación.

A igual que la clase ModbusQuery, implementa el patrón de diseño singleton para que la configuración de la aplicación pueda ser accedida desde toda la aplicación asegurando una instancia única.

Contiene dos tablas, modTable y plcTable. Estas tablas contienen la información que puede introducirse por el usuario desde la zona de configuración de la aplicación. En el momento en que se crea la instancia del objeto, este recoge los valores de la base de datos SQlite. Cada vez que el usuario cambia la configuración este objeto debe ser actualizado.

La labor de actualización y lectura de las tablas en la base de datos la facilita la propia clase mediante los siguientes métodos:

- getModTable() y get plcTable(): Obtienen las tablas de direcciones de registros a leer y conexiones respectivamente.
- addMod() y addPlc(): Estos métodos añaden una nueva entrada a la tabla.
- removeMod() y removePlc(): Eliminan una entrada de la base de datos.
- updateMod() y updatePlc(): Modifican una entrada determinada en la base de datos.

Referencias

[1] Definición IEC 61131.

[2] Mandado y otros.

[3] Kaschel y Pinto

[4] http://fieldbus.wikispaces.com/profibus

[5] http://fieldbus.wikispaces.com/profibus

[6] Profibus (PA/DP/FMS), Dto. de Tecnología Electrónica, UPCT http://es.scribd.com/doc/133252538/profibus-teoria

[7] Kaschel y Pinto

[8] Fichero main.c (anexo x)

Bibliografía

BALCELLS, Josep y José Luis ROMERAL: *Autómatas programables*, Ed. Marcombo, 1997.

KASCHEL, Héctor y Ernesto PINTO: *Análisis del estado del arte de los buses de campo aplicados al control de procesos industriales,* Universidad de Santiago de Chile, Fac. de Ingeniería Electrónica.

MANDADO, Enrique, Jorge Marcos ACEVEDO, Celso FERNÁNDEZ y José I. ARMESTO: *Autómatas programables y sistemas de automatización*, Barcelona, Ed. Marcombo, 2009 (2ª edición).

MORCILLO, Pedro y Julián CÓCERA: *Comunicaciones Industriales*, Ed. Paraninfo., 2000.

http://bues.ch/cms/hacking/profibus.html

http://www.chipkin.com

http://www.profibus.com

http://www.microchip.com

http://www.modbus.org

Anexo 1

Ficheros *TCPIP_MRF24W.c, HWP_MRF24W_XC32, MainDemo.c, MODBUSTCPServer.c y AppIOtest.c* de la aplicación servidor ModbusTCP.

AppIotest.c

```
1.
2. #include "TCPIP Stack/TCPIP.h"
3.
4.
5. extern WORD HOLDING_REG[20];
6.
7. static DWORD BlinkTimeLed1 = 0;
8. static DWORD BlinkTimeLed2 = 0;
9.
10.
11. void IO_Pull_Task(void){
12.
13.   static enum _sm_pulsador{
14.   wait_low = 0,
15.   wait_hi
16. }sm_pulsador=wait_low;
17.
18. ((LED0_IO)?(HOLDING_REG[0]=1):(HOLDING_REG[0]=0));
19. ((LED1_IO)?(HOLDING_REG[1]=1):(HOLDING_REG[1]=0));
20. ((LED2_IO)?(HOLDING_REG[2]=1):(HOLDING_REG[2]=0));
21.
22. switch(sm_pulsador){
23.   case wait_low:
24.         if(SW0_IO)break;
25.         sm_pulsador=wait_hi;
26.   break;
27. case wait_hi:
28.   if(!SW0_IO)break;
29.   HOLDING_REG[3]++;
30.   sm_pulsador=wait_low;
31. break;
32. }
33.
36.
37. ((SW0_IO)?(HOLDING_REG[6]=1):(HOLDING_REG[6]=0));
38.
39.
40.
41. if((TickGet() - BlinkTimeLed1) >= TICK_SECOND * (DWORD)HOLDING_REG[7]){
42.   LED1_IO = !LED1_IO;
43.   BlinkTimeLed1 = TickGet();
```

```
44. }
45.
46. if((TickGet() - BlinkTimeLed2) >= TICK_SECOND * (DWORD)HOLDING_REG[8]){
47.   LED2_IO = !LED2_IO;
48.   BlinkTimeLed2 = TickGet();
49. }
50.
51. }
52.
53.
54. void IO_Pull_Init(void){
55.
56.   HOLDING_REG[0] = 0;
57.   HOLDING_REG[1] = 0;
58.   HOLDING_REG[2] = 0;
59.   HOLDING_REG[3] = 0;
60.   HOLDING_REG[4] = 0;
61.   HOLDING_REG[5] = 0;
62.   HOLDING_REG[6] = 0;
63.   HOLDING_REG[7] = 2;
64.   HOLDING_REG[8] = 2;
65.
66. }
```

TCPIP MRF24W.h

```
1. /*********************************************************************
2. *
3. * Microchip TCP/IP Stack Demo Application Configuration Header
4. *
5. *********************************************************************
6. * FileName: TCPIPConfig.h
7. * Dependencies: Microchip TCP/IP Stack
8. * Processor: PIC32MX695F512H
9. * Compiler: Microchip XC32 Compiler
10. * Company: Microchip Technology, Inc.
11. *
12. * Software License Agreement
13. *
14. * Copyright (C) 2002-2013 Microchip Technology Inc. All rights
15. * reserved.
16. *
17. * Microchip licenses to you the right to use, modify, copy, and
18. * distribute:
19. * (i) the Software when embedded on a Microchip microcontroller or
20. * digital signal controller product ("Device") which is
21. * integrated into Licensee's product; or
22. * (ii) ONLY the Software driver source files ENC28J60.c, ENC28J60.h,
23. * ENCX24J600.c and ENCX24J600.h ported to a non-Microchip device
24. * used in conjunction with a Microchip ethernet controller for
25. * the sole purpose of interfacing with the ethernet controller.
26. *
```

```
27. * You should refer to the license agreement accompanying this
28. * Software for additional information regarding your rights and
29. * obligations.
30. *
31. * THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT
32. * WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT
33. * LIMITATION, ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A
34. * PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL
35. * MICROCHIP BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR
36. * CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF
37. * PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
38. * BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE
39. * THEREOF), ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER
40. * SIMILAR COSTS, WHETHER ASSERTED ON THE BASIS OF CONTRACT, TORT
41. * (INCLUDING NEGLIGENCE), BREACH OF WARRANTY, OR OTHERWISE.
42. *
43. *
44. ********************************************************************/
45. #ifndef __TCPIPCONFIG_H
46. #define __TCPIPCONFIG_H
47.
48. #include "GenericTypeDefs.h"
49. #include "Compiler.h"
50.
51. // ============================================================================
52. // Application Options
53. // ============================================================================
54.
55. /* Application Level Module Selection
56. * Uncomment or comment the following lines to enable or
57. * disabled the following high-level application modules.
58. */
59.
60. // Comments:
61. // iOS6.1 has wifi fixes that resolves connection issues with iPhone in softAP
mode.
62. //
63.
64. #define STACK_USE_MODBUS_TCP_SERVER
65. //#define STACK_USE_UART // Application demo using UART for IP address display and
stack configuration
66. //#define STACK_USE_UART2TCP_BRIDGE // UART to TCP Bridge application example
67. //#define STACK_USE_IP_GLEANING
68. #define STACK_USE_ICMP_SERVER // Ping query and response capability
69. //#define STACK_USE_ICMP_CLIENT // Ping transmission capability
70. #define STACK_USE_HTTP2_SERVER // New HTTP server with POST, Cookies,
Authentication, etc.
71. //#define STACK_USE_SSL_SERVER // SSL server socket support (Requires SW300052)
72. //#define STACK_USE_SSL_CLIENT // SSL client socket support (Requires SW300052)
73. //#define STACK_USE_AUTO_IP // Dynamic link-layer IP address automatic
configuration protocol
74. #define STACK_USE_DHCP_CLIENT // Dynamic Host Configuration Protocol client for
obtaining IP address and other parameters
75. #define STACK_USE_DHCP_SERVER // Single host DHCP server
76. //#define STACK_USE_FTP_SERVER // File Transfer Protocol (old)
77. //#define STACK_USE_SMTP_CLIENT // Simple Mail Transfer Protocol for sending email
```

```
78. //#define STACK_USE_SNMP_SERVER // Simple Network Management Protocol v2C Community
Agent
79. //#define STACK_USE_SNMPV3_SERVER // Simple Network Management Protocol v3 Agent
80. //#define STACK_USE_TFTP_CLIENT // Trivial File Transfer Protocol client
81. //#define STACK_USE_TELNET_SERVER // Telnet server
82. #define STACK_USE_ANNOUNCE // Microchip Embedded Ethernet Device Discoverer
server/client
83. #define STACK_USE_DNS // Domain Name Service Client for resolving hostname strings
to IP addresses
84. #define STACK_USE_DNS_SERVER // Domain Name Service Server for redirection to the
local device
85. #define STACK_USE_NBNS // NetBIOS Name Service Server for responding to NBNS
hostname broadcast queries
86. #define STACK_USE_REBOOT_SERVER // Module for resetting this PIC remotely.
Primarily useful for a Bootloader.
87. //#define STACK_USE_SNTP_CLIENT // Simple Network Time Protocol for obtaining
current date/time from Internet
88. //#define STACK_USE_UDP_PERFORMANCE_TEST // Module for testing UDP TX performance
characteristics. NOTE: Enabling this will cause a huge amount 89. //#define
STACK_USE_TCP_PERFORMANCE_TEST // Module for testing TCP TX performance characteristics
90. //#define STACK_USE_DYNAMICDNS_CLIENT // Dynamic DNS client updater module
91. //#define STACK_USE_BERKELEY_API // Berkerly Sockets APIs are available
92. #define STACK_USE_ZEROCONF_LINK_LOCAL // Zeroconf IPv4 Link-Local Addressing
93. #define STACK_USE_ZEROCONF_MDNS_SD // Zeroconf mDNS and mDNS service discovery
94.
95.
96. // ============================================================================
97. // Data Storage Options
98. // ============================================================================
99.
100. /* MPFS Configuration
101. * MPFS is automatically included when required for other
102. * applications. If your custom application requires it
103. * otherwise, uncomment the appropriate selection.
104. */
105. #define STACK_USE_MPFS2
106.
107. /* MPFS Storage Location
108. * If html pages are stored in internal program memory,
109. * comment both MPFS_USE_EEPROM and MPFS_USE_SPI_FLASH, then
110. * include an MPFS image (.c or .s file) in the project.
111. * If html pages are stored in external memory, uncomment the
112. * appropriate definition.
113. *
114. * Supported serial flash parts include the SST25VFxxxB series.
115. */
116. //#define MPFS_USE_EEPROM
117. //#define MPFS_USE_SPI_FLASH
118.
119. /* EEPROM Addressing Selection
120. * If using the 1Mbit EEPROM, uncomment this line
121. */
122. //#define USE_EEPROM_25LC1024
123.
124. /* EEPROM Reserved Area
125. * Number of EEPROM bytes to be reserved before MPFS storage starts.
```

```
126. * These bytes host application configurations such as IP Address,
127. * MAC Address, and any other required variables.
128. *
129. * For MPFS Classic, this setting must match the Reserved setting
130. * on the Advanced Settings page of the MPFS2 Utility.
131. */
132. #define MPFS_RESERVE_BLOCK (205ul)
133.
134. /* MPFS File Handles
135. * Maximum number of simultaneously open MPFS2 files.
136. * For MPFS Classic, this has no effect.
137. */
138. #define MAX_MPFS_HANDLES (7ul)
139.
140.
141. // ============================================================================
142. // Network Addressing Options
143. // ============================================================================
144.
145. /* Default Network Configuration
146. * These settings are only used if data is not found in EEPROM.
147. * To clear EEPROM, hold BUTTON0, reset the board, and continue
148. * holding until the LEDs flash. Release, and reset again.
149. */
150. #define MY_DEFAULT_HOST_NAME "MCHPBOARD"
151.
152. #define MY_DEFAULT_MAC_BYTE1 (0x00) // Use the default of 00-04-A3-00-00-00
153. #define MY_DEFAULT_MAC_BYTE2 (0x04) // if using an ENCX24J600, MRF24WB0M, or
154. #define MY_DEFAULT_MAC_BYTE3 (0xA3) // PIC32MX6XX/7XX internal Ethernet
155. #define MY_DEFAULT_MAC_BYTE4 (0x00) // controller and wish to use the
156. #define MY_DEFAULT_MAC_BYTE5 (0x00) // internal factory programmed MAC
157. #define MY_DEFAULT_MAC_BYTE6 (0x00) // address instead.
158.
159. #define MY_DEFAULT_IP_ADDR_BYTE1 (192ul)
160. #define MY_DEFAULT_IP_ADDR_BYTE2 (128ul)
161. #define MY_DEFAULT_IP_ADDR_BYTE3 (1ul)
162. #define MY_DEFAULT_IP_ADDR_BYTE4 (3ul)
163.
164. #define MY_DEFAULT_GATE_BYTE1 (192ul)
165. #define MY_DEFAULT_GATE_BYTE2 (128ul)
166. #define MY_DEFAULT_GATE_BYTE3 (1ul)
167. #define MY_DEFAULT_GATE_BYTE4 (3ul)
168.
169. #define MY_DEFAULT_PRIMARY_DNS_BYTE1 (192ul)
170. #define MY_DEFAULT_PRIMARY_DNS_BYTE2 (128ul)
171. #define MY_DEFAULT_PRIMARY_DNS_BYTE3 (1ul)
172. #define MY_DEFAULT_PRIMARY_DNS_BYTE4 (3ul)
173.
174. #define MY_DEFAULT_MASK_BYTE1 (255ul)
175. #define MY_DEFAULT_MASK_BYTE2 (255ul)
176. #define MY_DEFAULT_MASK_BYTE3 (255ul)
177. #define MY_DEFAULT_MASK_BYTE4 (0ul)
178.
179.
180. #define MY_DEFAULT_SECONDARY_DNS_BYTE1 (0ul)
181. #define MY_DEFAULT_SECONDARY_DNS_BYTE2 (0ul)
```

```
182. #define MY_DEFAULT_SECONDARY_DNS_BYTE3 (0ul)
183. #define MY_DEFAULT_SECONDARY_DNS_BYTE4 (0ul)
184.
185. // ============================================================================
186. // PIC32MX695F512H MAC Layer Options
187. // If not using a PIC32MX7XX/6XX device, ignore this section.
188. // ============================================================================
189. #define ETH_CFG_LINK 0 // set to 1 if you need to config the link to specific
following parameters
190. // otherwise the default connection will be attempted
191. // depending on the selected PHY
192. #define ETH_CFG_AUTO 1 // use auto negotiation
193. #define ETH_CFG_10 1 // use/advertise 10 Mbps capability
194. #define ETH_CFG_100 1 // use/advertise 100 Mbps capability
195. #define ETH_CFG_HDUPLEX 1 // use/advertise half duplex capability
196. #define ETH_CFG_FDUPLEX 1 // use/advertise full duplex capability
197. #define ETH_CFG_AUTO_MDIX 1 // use/advertise auto MDIX capability
198. #define ETH_CFG_SWAP_MDIX 1 // use swapped MDIX. else normal MDIX
199.
200. #define EMAC_TX_DESCRIPTORS 2 // number of the TX descriptors to be created
201. #define EMAC_RX_DESCRIPTORS 8 // number of the RX descriptors and RX buffers to be
created
202.
203. #define EMAC_RX_BUFF_SIZE 1536 // size of a RX buffer. should be multiple of 16
204. // this is the size of all receive buffers processed by the ETHC
205. // The size should be enough to accomodate any network received packet
206. // If the packets are larger, they will have to take multiple RX buffers
207. // The current implementation does not handle this situation right now and the
packet is discarded.
208.
209.
210. // ============================================================================
211. // Transport Layer Options
212. // ============================================================================
213.
214. /* Transport Layer Configuration
215. * The following low level modules are automatically enabled
216. * based on module selections above. If your custom module
217. * requires them otherwise, enable them here.
218. */
219. #define STACK_USE_TCP
220. #define STACK_USE_UDP
221.
222. /* Client Mode Configuration
223. * Uncomment following line if this stack will be used in CLIENT
224. * mode. In CLIENT mode, some functions specific to client operation
225. * are enabled.
226. */
227. #define STACK_CLIENT_MODE
228.
229. /* TCP Socket Memory Allocation
230. * TCP needs memory to buffer incoming and outgoing data. The
231. * amount and medium of storage can be allocated on a per-socket
232. * basis using the example below as a guide.
233. */
234. // Allocate how much total RAM (in bytes) you want to allocate
```

```
235. // for use by your TCP TCBs, RX FIFOs, and TX FIFOs.
236. #define TCP_ETH_RAM_SIZE (8192ul)
237. #define TCP_PIC_RAM_SIZE (0ul)
238. #define TCP_SPI_RAM_SIZE (0ul)
239. #define TCP_SPI_RAM_BASE_ADDRESS (0x00)
240.
241. // Define names of socket types
242. #define TCP_SOCKET_TYPES
243. #define TCP_PURPOSE_GENERIC_TCP_CLIENT 0
244. #define TCP_PURPOSE_GENERIC_TCP_SERVER 1
245. #define TCP_PURPOSE_TELNET 2
246. #define TCP_PURPOSE_FTP_COMMAND 3
247. #define TCP_PURPOSE_FTP_DATA 4
248. #define TCP_PURPOSE_TCP_PERFORMANCE_TX 5
249. #define TCP_PURPOSE_TCP_PERFORMANCE_RX 6
250. #define TCP_PURPOSE_UART_2_TCP_BRIDGE 7
251. #define TCP_PURPOSE_HTTP_SERVER 8
252. #define TCP_PURPOSE_DEFAULT 9
253. #define TCP_PURPOSE_BERKELEY_SERVER 10
254. #define TCP_PURPOSE_BERKELEY_CLIENT 11
255. #define TCP_PURPOSE_MODBUS_TCP_SERVER 12
256. #define END_OF_TCP_SOCKET_TYPES
257.
258. #if defined(__TCP_C)
259. // Define what types of sockets are needed, how many of
260. // each to include, where their TCB, TX FIFO, and RX FIFO
261. // should be stored, and how big the RX and TX FIFOs should
262. // be. Making this initializer bigger or smaller defines
263. // how many total TCP sockets are available.
264. //
265. // Each socket requires up to 56 bytes of PIC RAM and
266. // 48+(TX FIFO size)+(RX FIFO size) bytes of TCP_*_RAM each.
267. //
268. // Note: The RX FIFO must be at least 1 byte in order to
269. // receive SYN and FIN messages required by TCP. The TX
270. // FIFO can be zero if desired.
271. #define TCP_CONFIGURATION
272. ROM struct
273. {
274.   BYTE vSocketPurpose;
275.   BYTE vMemoryMedium;
276.   WORD wTXBufferSize;
277.   WORD wRXBufferSize;
278. } TCPSocketInitializer[] =
279. {
280. //{TCP_PURPOSE_GENERIC_TCP_CLIENT, TCP_ETH_RAM, 2048, 20},
281. //{TCP_PURPOSE_GENERIC_TCP_SERVER, TCP_ETH_RAM, 20, 2048},
282. {TCP_PURPOSE_TELNET, TCP_ETH_RAM, 200, 150},
283. //{TCP_PURPOSE_TELNET, TCP_ETH_RAM, 200, 150},
284. //{TCP_PURPOSE_TELNET, TCP_ETH_RAM, 200, 150},
285. //{TCP_PURPOSE_FTP_COMMAND, TCP_ETH_RAM, 100, 40},
286. //{TCP_PURPOSE_FTP_DATA, TCP_ETH_RAM, 0, 128},
287. //{TCP_PURPOSE_TCP_PERFORMANCE_TX, TCP_ETH_RAM, 200, 1},
288. //{TCP_PURPOSE_TCP_PERFORMANCE_RX, TCP_ETH_RAM, 40, 1500},
289. //{TCP_PURPOSE_UART_2_TCP_BRIDGE, TCP_ETH_RAM, 256, 256},
290. {TCP_PURPOSE_HTTP_SERVER, TCP_ETH_RAM, 2048, 1000},
```

```
291. {TCP_PURPOSE_HTTP_SERVER, TCP_ETH_RAM, 2048, 1000},
292. //{TCP_PURPOSE_DEFAULT, TCP_ETH_RAM, 1000, 1000},
293. //{TCP_PURPOSE_BERKELEY_SERVER, TCP_ETH_RAM, 25, 20},
294. //{TCP_PURPOSE_BERKELEY_SERVER, TCP_ETH_RAM, 25, 20},
295. //{TCP_PURPOSE_BERKELEY_SERVER, TCP_ETH_RAM, 25, 20},
296. //{TCP_PURPOSE_BERKELEY_CLIENT, TCP_ETH_RAM, 125, 100},
297. {TCP_PURPOSE_MODBUS_TCP_SERVER, TCP_ETH_RAM, 150, 150},
298. };
299. #define END_OF_TCP_CONFIGURATION
300. #endif
301.
302. /* UDP Socket Configuration
303. * Define the maximum number of available UDP Sockets, and whether
304. * or not to include a checksum on packets being transmitted.
305. */
306. #define MAX_UDP_SOCKETS (8u)
307. //#define UDP_USE_TX_CHECKSUM // This slows UDP TX performance by nearly 50%,
except when using the ENCX24J600 or PIC32MX6XX/7XX, which 308.
309.
310. /* Berkeley API Sockets Configuration
311. * Note that each Berkeley socket internally uses one TCP or UDP socket
312. * defined by MAX_UDP_SOCKETS and the TCPSocketInitializer[] array.
313. * Therefore, this number MUST be less than or equal to MAX_UDP_SOCKETS + the
314. * number of TCP sockets defined by the TCPSocketInitializer[] array
315. * (i.e. sizeof(TCPSocketInitializer)/sizeof(TCPSocketInitializer[0])).
316. * This define has no effect if STACK_USE_BERKELEY_API is not defined and
317. * Berkeley Sockets are disabled. Set this value as low as your application
318. * requires to avoid waisting RAM.
319. */
320. #define BSD_SOCKET_COUNT (5u)
321.
322.
323. // ==========================================================================
324. // Application-Specific Options
325. // ==========================================================================
326.
327. // -- HTTP2 Server options ---------------------------------------------
328.
329. // Maximum numbers of simultaneous HTTP connections allowed.
330. // Each connection consumes 2 bytes of RAM and a TCP socket
331. #define MAX_HTTP_CONNECTIONS (2u)
332.
333. // Optional setting to use PIC RAM instead of Ethernet/Wi-Fi RAM for
334. // storing HTTP Connection Context variables (HTTP_CONN structure for each
335. // HTTP connection). Undefining this macro results in the Ethernet/Wi-Fi
336. // RAM being used (minimum PIC RAM usage, lower performance). Defining
337. // this macro results in PIC RAM getting used (higher performance, but uses
338. // PIC RAM). This option should not be enabled on PIC18 devices. The
339. // performance increase of having this option defined is only apparent when
340. // the HTTP server is servicing multiple connections simultaneously.
341. //#define HTTP_SAVE_CONTEXT_IN_PIC_RAM
342.
343. // Indicate what file to serve when no specific one is requested
344. #define HTTP_DEFAULT_FILE "index.htm"
345. #define HTTPS_DEFAULT_FILE "index.htm"
346. #define HTTP_DEFAULT_LEN (10u) // For buffer overrun protection.
```

```
347. // Set to longest length of above two strings.
348.
349. // Configure MPFS over HTTP updating
350. // Comment this line to disable updating via HTTP
351. #define HTTP_MPFS_UPLOAD "mpfsupload"
352. //#define HTTP_MPFS_UPLOAD_REQUIRES_AUTH // Require password for MPFS uploads
353. // Certain firewall and router combinations cause the MPFS2 Utility to fail
354. // when uploading. If this happens, comment out this definition.
355.
356. // Define which HTTP modules to use
357. // If not using a specific module, comment it to save resources
358. #define HTTP_USE_POST // Enable POST support
359. //#define HTTP_USE_COOKIES // Enable cookie support
360. //#define HTTP_USE_AUTHENTICATION // Enable basic authentication support
361.
362. //#define HTTP_NO_AUTH_WITHOUT_SSL // Uncomment to require SSL before requesting a
password
363.
364. // Define the listening port for the HTTP server
365. #define HTTP_PORT (80u)
366.
367. // Define the listening port for the HTTPS server (if STACK_USE_SSL_SERVER is
enabled)
368. #define HTTPS_PORT (443u)
369.
370. // Define the maximum data length for reading cookie and GET/POST arguments
(bytes)
371. #define HTTP_MAX_DATA_LEN (100u)
372.
373. // Define the minimum number of bytes free in the TX FIFO before executing
callbacks
374. #define HTTP_MIN_CALLBACK_FREE (1024u)
375.
376. //#define STACK_USE_HTTP_APP_RECONFIG // Use the AppConfig web page in the Demo
App (~2.5kb ROM, ~0b RAM)
377. //#define STACK_USE_HTTP_MD5_DEMO // Use the MD5 Demo web page (~5kb ROM, ~160b
RAM)
378. //#define STACK_USE_HTTP_EMAIL_DEMO // Use the e-mail demo web page
379.
380. // -- SSL Options ----------------------------------------------------------
381.
382. #define MAX_SSL_CONNECTIONS (2ul) // Maximum connections via SSL
383. #define MAX_SSL_SESSIONS (2ul) // Max # of cached SSL sessions
384. #define MAX_SSL_BUFFERS (4ul) // Max # of SSL buffers (2 per socket)
385. #define MAX_SSL_HASHES (5ul) // Max # of SSL hashes (2 per, plus 1 to avoid
deadlock)
386.
387. // Bits in SSL RSA key. This parameter is used for SSL sever
388. // connections only. The only valid value is 512 bits (768 and 1024
389. // bits do not work at this time). Note, however, that SSL client
390. // operations do currently work up to 1024 bit RSA key length.
391. #define SSL_RSA_KEY_SIZE (512ul)
392.
393.
394. // -- Telnet Options -------------------------------------------------------
395.
```

```
396. // Number of simultaneously allowed Telnet sessions. Note that you
397. // must have an equal number of TCP_PURPOSE_TELNET type TCP sockets
398. // declared in the TCPSocketInitializer[] array above for multiple
399. // connections to work. If fewer sockets are available than this
400. // definition, then the the lesser of the two quantities will be the
401. // actual limit.
402. #define MAX_TELNET_CONNECTIONS (1u)
403.
404. // Default local listening port for the Telnet server. Port 23 is the
405. // protocol default.
406. #define TELNET_PORT 23
407.
408. // Default local listening port for the Telnet server when SSL secured.
409. // Port 992 is the telnets protocol default.
410. #define TELNETS_PORT 992
411.
412. // Force all connecting clients to be SSL secured and connected via
413. // TELNETS_PORT. Connections on port TELNET_PORT will be ignored. If
414. // STACK_USE_SSL_SERVER is undefined, this entire setting is ignored
415. // (server will accept unsecured connections on TELNET_PORT and won't even
416. // listen on TELNETS_PORT).
417. //#define TELNET_REJECT_UNSECURED
418.
419. // Default username and password required to login to the Telnet server.
420. #define TELNET_USERNAME "admin"
421. #define TELNET_PASSWORD "microchip"
422.
423. #endif
```

MainDemo.c

```
1. /*********************************************************************
2. *
3. * Main Application Entry Point
4. * -Demonstrates how to call and use the Microchip WiFi Module and
5. * TCP/IP stack
6. * -Reference: Microchip TCP/IP Stack Help (TCPIP Stack Help.chm)
7. *
8. *********************************************************************
9. * FileName: MainDemo.c
10. * Dependencies: TCPIP.h
11. * Processor: PIC32MX695F512H
12. * Compiler: Microchip XC32 Compiler
13. * Company: Microchip Technology, Inc.
14. *
15. * Software License Agreement
16. *
17. * Copyright (C) 2002-2013 Microchip Technology Inc. All rights
18. * reserved.
19. *
20. * Microchip licenses to you the right to use, modify, copy, and
21. * distribute:
22. * (i) the Software when embedded on a Microchip microcontroller or
```

```
23. * digital signal controller product ("Device") which is
24. * integrated into Licensee's product; or
25. * (ii) ONLY the Software driver source files ENC28J60.c, ENC28J60.h,
26. * ENCX24J600.c and ENCX24J600.h ported to a non-Microchip device
27. * used in conjunction with a Microchip ethernet controller for
28. * the sole purpose of interfacing with the ethernet controller.
29. *
30. * You should refer to the license agreement accompanying this
31. * Software for additional information regarding your rights and
32. * obligations.
33. *
34. * THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT
35. * WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT
36. * LIMITATION, ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A
37. * PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL
38. * MICROCHIP BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR
39. * CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF
40. * PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
41. * BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE
42. * THEREOF), ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER
43. * SIMILAR COSTS, WHETHER ASSERTED ON THE BASIS OF CONTRACT, TORT
44. * (INCLUDING NEGLIGENCE), BREACH OF WARRANTY, OR OTHERWISE.
45. *
46. * File Description:
47. * Change History:
48. * Date Comment
49. * ---------- ----------------------------------------
50. * 10/22/2012 Initial release Wifi G Demo (MRF24WG)
51. ********************************************************************/
52. /*
53. * This macro uniquely defines this file as the main entry point.
54. * There should only be one such definition in the entire project,
55. * and this file must define the AppConfig variable as described below.
56. */
57. #define THIS_IS_STACK_APPLICATION
58.
59. #include "MainDemo.h"
60. #include "AppIOtest.h"
61. #include "MODBUSTCPServer.h"
62.
63. #if defined( WF_CONSOLE )
64. #include "TCPIP Stack/WFConsole.h"
65. #include "IperfApp.h"
66. #endif
67.
68. //
69. // Differences to wifi comm demo board (MRF24WB0MA) :
70. // Wifi comm demo board is centered on variable CPElements.
71. // Wifi comm demo SSID : MCHP_xxxx
72. // SW0 functions : On powerup initiates self test.
73. //
74. // Wifi G demo board is centered on variable AppConfig, since this is the generic
approach adopted by
75. // TCPIP demo/ezconsole/ezconfig apps.
76. // Wifi G demo SSID : MCHP_G_xxxx
77. // SW0 functions : On powerup initiates self test.
```

```
78. // When running, initiates reboot to factory default conditions.
79. //
80.
81. //
82. // Wifi G Demo Web Pages
83. // Generate a WifiGDemoMPFSImg.c file using the MPFS utility (refer to Convert
WebPages to MPFS.bat)
84. // that gets compiled into source code and programmed into the flash of the uP.
85.
86. APP_CONFIG AppConfig;
87.
88. static unsigned short wOriginalAppConfigChecksum; // Checksum of the ROM defaults
for AppConfig
89. extern unsigned char TelnetPut(unsigned char c);
90.
91. UINT8 g_scan_done = 0; // WF_PRESCAN This will be set wheneven event scan results
are ready.
92. UINT8 g_prescan_waiting = 1; // WF_PRESCAN This is used only to allow POR prescan
once.
93.
94. // Private helper functions.
95. static void InitAppConfig(void);
96. static void InitializeBoard(void);
97. static void SelfTest(void);
98. extern void WF_Connect(void);
99. void UARTTxBuffer(char *buffer, UINT32 size);
100.
101.
102. // Used for re-directing printf and UART statements to the Telnet daemon
103. void _mon_putc(char c) {
104.
105.   #ifdef STACK_USE_TELNET_SERVER
106.   TelnetPut(c);
107.   #endif
108. }
109.
110. // Exception Handlers
111. // If your code gets here, you either tried to read or write
112. // a NULL pointer, or your application overflowed the stack
113. // by having too many local variables or parameters declared.
114. void _general_exception_handler(unsigned cause, unsigned status)
115. {
116.   Nop();
117.   Nop();
118. }
119.
120. // ************************************************************
121. // Main application entry point.
122. // ************************************************************
123. int main(void)
124. {
125.   static DWORD t = 0;
126.   static DWORD dwLastIP = 0;
127.   #if defined (EZ_CONFIG_STORE)
128.   static DWORD ButtonPushStart = 0;
129.   #endif
```

```
130.   UINT8 channelList[] = MY_DEFAULT_CHANNEL_LIST_PRESCAN; // WF_PRESCAN
131.   tWFScanResult bssDesc;
132.   #if 0
133.   INT8 TxPower; // Needed to change MRF24WG transmit power.
134.   #endif
135.
136. // Initialize application specific hardware
137.   InitializeBoard();
138.
139. // Initialize TCP/IP stack timer
140.   TickInit();
141.
142. #if defined(STACK_USE_MPFS2)
143. // Initialize the MPFS File System
144. // Generate a WifiGDemoMPFSImg.c file using the MPFS utility (refer to Convert
WebPages to MPFS.bat)
145. // that gets compiled into source code and programmed into the flash of the uP.
146.   MPFSInit();
147.   #endif
148.
149. // Initialize Stack and application related NV variables into AppConfig.
150.   InitAppConfig();
151.
152. // Initialize core stack layers (MAC, ARP, TCP, UDP) and
153. // application modules (HTTP, SNMP, etc.)
154.   StackInit();
155.
156.   #if 0
157. // Below is used to change MRF24WG transmit power.
158. // This has been verified to be functional (Jan 2013)
159.   if (AppConfig.networkType == WF_SOFT_AP)
160. {
161.   WF_TxPowerGetMax(&TxPower);
162.   WF_TxPowerSetMax(TxPower);
163. }
164. #endif
165.
166. // Run Self Test if SW0 pressed on startup
167. if(SW0_IO == 0)
168.   SelfTest();
169.
170. #ifdef STACK_USE_TELNET_SERVER
171. // Initialize Telnet and
172. // Put Remote client in Remote Character Echo Mode
173. TelnetInit();
174. putc(0xff, stdout); // IAC = Interpret as Command
175. putc(0xfe, stdout); // Type of Operation = DONT
176. putc(0x22, stdout); // Option = linemode
177. putc(0xff, stdout); // IAC = Interpret as Command
178. putc(0xfb, stdout); // Type of Operation = DO
179. putc(0x01, stdout); // Option = echo
180. #endif
181.
182.
183. #if defined ( EZ_CONFIG_SCAN )
184. // Initialize WiFi Scan State Machine NV variables
```

```
185. WFInitScan();
186. #endif
187.
188. // WF_PRESCAN: Pre-scan before starting up as SoftAP mode
189. WF_CASetScanType(MY_DEFAULT_SCAN_TYPE);
190. WF_CASetChannelList(channelList, sizeof(channelList));
191.
192. if (WFStartScan() == WF_SUCCESS) {
193.   CAN_SET_DISPLAY(SCANCXT.scanState);
194.   SCANCXT.displayIdx = 0;
195. }
196.
197. // Needed to trigger g_scan_done
198. WFRetrieveScanResult(0, &bssDesc);
199.
200. #if defined(STACK_USE_ZEROCONF_LINK_LOCAL)
201. // Initialize Zeroconf Link-Local state-machine, regardless of network type.
202. ZeroconfLLInitialize();
203. #endif
204.
205. IO_Pull_Init();
206.
207. #if defined(STACK_USE_ZEROCONF_MDNS_SD)
208. // Initialize DNS Host-Name from TCPIPConfig.h, regardless of network type.
209. mDNSInitialize(MY_DEFAULT_HOST_NAME);
210. mDNSServiceRegister(
211. // (const char *) AppConfig.NetBIOSName, // base name of the service. Ensure
uniformity with CheckHibernate().
212. (const char *) "DemoWebServer", // base name of the service. Ensure uniformity
with CheckHibernate().
213. "_http._tcp.local", // type of the service
214. 80, // TCP or UDP port, at which this service is available
215. ((const BYTE *)"path=/index.htm"), // TXT info
216. 1, // auto rename the service when if needed
217. NULL, // no callback function
218. NULL // no application context
219. );
220. mDNSMulticastFilterRegister();
221. #endif
222.
223. #if defined(WF_CONSOLE)
224. // Initialize the WiFi Console App
225. WFConsoleInit();
226. #endif
227.
228. // Now that all items are initialized, begin the co-operative
229. // multitasking loop. This infinite loop will continuously
230. // execute all stack-related tasks, as well as your own
231. // application's functions. Custom functions should be added
232. // at the end of this loop.
233. // Note that this is a "co-operative mult-tasking" mechanism
234. // where every task performs its tasks (whether all in one shot
235. // or part of it) and returns so that other tasks can do their
236. // job.
237. // If a task needs very long time to do its job, it must be broken
238. // down into smaller pieces so that other tasks can have CPU time.
```

```
239. while(1)
240. {
241.  if (AppConfig.networkType == WF_SOFT_AP) {
242.         if (g_scan_done) {
243.                if (g_prescan_waiting) {
244.                       SCANCXT.displayIdx = 0;
245.                       while (IS_SCAN_STATE_DISPLAY(SCANCXT.scanState)) {
246.                              WFDisplayScanMgr();
247.                       }
248.
249.                       #if defined(WF_CS_TRIS)
250.                       WF_Connect();
251.                       #endif
252.                       g_scan_done = 0;
253.                       g_prescan_waiting = 0;
254.                }
255.         }
256.  }
257.
258. #if defined (EZ_CONFIG_STORE)
259. // Hold SW0 for 4 seconds to reset to defaults.
260. if (SW0_IO == 0u) { // Button is pressed
261.  if (ButtonPushStart == 0) //Just pressed
262.         ButtonPushStart = TickGet();
263.  else
264.  if(TickGet() - ButtonPushStart > 4*TICK_SECOND)
265.         RestoreWifiConfig();
266.  }
267.   else
268. {
269.  ButtonPushStart = 0; //Button release reset the clock
270. }
271.
272. if (AppConfig.saveSecurityInfo)
273. {
274. // set true by WF_ProcessEvent after connecting to a new network
275. // get the security info, and if required, push the PSK to EEPROM
276.  if ((AppConfig.SecurityMode == WF_SECURITY_WPA_WITH_PASS_PHRASE) ||
277. (AppConfig.SecurityMode == WF_SECURITY_WPA2_WITH_PASS_PHRASE) ||
278. (AppConfig.SecurityMode == WF_SECURITY_WPA_AUTO_WITH_PASS_PHRASE))
279. {
280. // only need to save when doing passphrase
281. tWFCPElements profile;
282. UINT8 connState;
283. UINT8 connID;
284. WF_CMGetConnectionState(&connState, &connID);
285. WF_CPGetElements(connID, &profile);
286.
287. memcpy((char*)AppConfig.SecurityKey, (char*)profile.securityKey, 32);
288. AppConfig.SecurityMode--; // the calc psk is exactly one below for each passphrase
option
289. AppConfig.SecurityKeyLength = 32;
290.
291. SaveAppConfig(&AppConfig);
292. }
293.
```

```
294. AppConfig.saveSecurityInfo = FALSE;
295. }
296. #endif // EZ_CONFIG_STORE
297.
298. // Blink LED0 twice per sec when unconfigured, once per sec after config
299. if((TickGet() - t >= TICK_SECOND/(4ul - (CFGCXT.isWifiDoneConfigure*2ul))))
300. {
301.   t = TickGet();
302.   LED0_INV();
303. }
304.
305. // This task performs normal stack task including checking
306. // for incoming packet, type of packet and calling
307. // appropriate stack entity to process it.
308. StackTask();
309.
310. // This task invokes each of the core stack application tasks
311. StackApplications();
312.
313. // Enable WF_USE_POWER_SAVE_FUNCTIONS
314. WiFiTask();
315.
316. #if defined(STACK_USE_ZEROCONF_LINK_LOCAL)
317. ZeroconfLLProcess();
318. #endif
319.
320. #if defined(STACK_USE_ZEROCONF_MDNS_SD)
321. mDNSProcess();
322. #endif
323.
324. IO_Pull_Task();
325.
326. #if defined(STACK_USE_MODBUS_TCP_SERVER)
327. MODBUSTCPServer();
328. #endif
329.
330. // Process application specific tasks here.
331. // Any custom modules or processing you need to do should
332. // go here.
333. #if defined(WF_CONSOLE)
334. WFConsoleProcess();
335. WFConsoleProcessEpilogue();
336. #endif
337.
338. // If the local IP address has changed (ex: due to DHCP lease change)
339. // write the new IP address to the LCD display, UART, and Announce
340. // service
341. if(dwLastIP != AppConfig.MyIPAddr.Val)
342. {
343. dwLastIP = AppConfig.MyIPAddr.Val;
344. DisplayIPValue(AppConfig.MyIPAddr);
345.
346. #if defined(STACK_USE_ANNOUNCE)
347.   AnnounceIP();
348. #endif
349.
```

```
350. #if defined(STACK_USE_ZEROCONF_MDNS_SD)
351.   mDNSFillHostRecord();
352. #endif
353. }
354.
355. }
356. }
357.
358. /*****************************************************************************
359. Function:
360. static void InitializeBoard(void)
361.
362. Description:
363. This routine initializes the hardware. It is a generic initialization
364. routine for many of the Microchip development boards, using definitions
365. in HardwareProfile.h to determine specific initialization.
366.
367. Precondition:
368. None
369.
370. Parameters:
371. None - None
372.
373. Returns:
374. None
375.
376. Remarks:
377. None
378. *****************************************************************************/
379. static void InitializeBoard(void)
380. {
381. // Note: WiFi Module hardware Initialization handled by StackInit() Library
Routine
382.
383. // Enable multi-vectored interrupts
384. INTEnableSystemMultiVectoredInt();
385.
386. // Enable optimal performance
387. SYSTEMConfigPerformance(GetSystemClock());
388.
389. // Use 1:1 CPU Core:Peripheral clocks
390. mOSCSetPBDIV(OSC_PB_DIV_1);
391.
392. // Disable JTAG port so we get our I/O pins back, but first
393. // wait 50ms so if you want to reprogram the part with
394. // JTAG, you'll still have a tiny window before JTAG goes away.
395. // The PIC32 Starter Kit debuggers use JTAG and therefore must not
396. // disable JTAG.
397. DelayMs(50);
398. DDPCONbits.JTAGEN = 0;
399.
400. // LEDs
401. LEDS_OFF();
402. LED0_TRIS = 0;
403. LED1_TRIS = 0;
404. LED2_TRIS = 0;
```

```
405.
406. // Push Button
407. SW0_TRIS = 1;
408.
409. }
410.
411. static ROM BYTE SerializedMACAddress[6] = {MY_DEFAULT_MAC_BYTE1,
MY_DEFAULT_MAC_BYTE2, MY_DEFAULT_MAC_BYTE3, MY_DEFAULT_MAC_BYTE4, MY_DEFAULT_412.
413. /*********************************************************************
414. * Function: void InitAppConfig(void)
415. *
416. * PreCondition: MPFSInit() is already called.
417. *
418. * Input: None
419. *
420. * Output: Write/Read non-volatile config variables.
421. *
422. * Side Effects: None
423. *
424. * Overview: None
425. *
426. * Note: None
427. ********************************************************************/
428. static void InitAppConfig(void)
429. {
430. #if defined(EEPROM_CS_TRIS) || defined(SPIFLASH_CS_TRIS)
431. unsigned char vNeedToSaveDefaults = 0;
432. #endif
433.
434. while(1)
435. {
436. // Start out zeroing all AppConfig bytes to ensure all fields are
437. // deterministic for checksum generation
438. memset((void*)&AppConfig, 0x00, sizeof(AppConfig));
439.
440. AppConfig.Flags.bIsDHCPEnabled = TRUE;
441. AppConfig.Flags.bInConfigMode = TRUE;
442. memcpypgm2ram((void*)&AppConfig.MyMACAddr, (ROM void*)SerializedMACAddress,
sizeof(AppConfig.MyMACAddr));
443. // {
444. // _prog_addressT MACAddressAddress;
445. // MACAddressAddress.next = 0x157F8;
446. // _memcpy_p2d24((char*)&AppConfig.MyMACAddr, MACAddressAddress,
sizeof(AppConfig.MyMACAddr));
447. // }
448.
449.
450. // SoftAP on certain setups with IP 192.168.1.1 has problem with DHCP client
assigning new IP address on redirection.
451. // 192.168.1.1 is a common IP address with most APs. This is still under
investigation.
452. // For now, assign this as 192.168.1.3
453.
454. AppConfig.MyIPAddr.Val = 192ul | 168ul<<8ul | 1ul<<16ul | 3ul<<24ul;
455. AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
456. AppConfig.MyMask.Val = 255ul | 255ul<<8ul | 0ul<<16ul | 0ul<<24ul;
```

```
457.  AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
458.  AppConfig.MyGateway.Val = AppConfig.MyIPAddr.Val;
459.  AppConfig.PrimaryDNSServer.Val = AppConfig.MyIPAddr.Val;
460.  AppConfig.SecondaryDNSServer.Val = AppConfig.MyIPAddr.Val;
461.
462.  // Load the default NetBIOS Host Name
463.  memcpypgm2ram(AppConfig.NetBIOSName, (ROM void*)MY_DEFAULT_HOST_NAME, 16);
464.  FormatNetBIOSName(AppConfig.NetBIOSName);
465.
466.  #if defined(WF_CS_TRIS)
467.  // Load the default SSID Name
468.  WF_ASSERT(sizeof(MY_DEFAULT_SSID_NAME)-1 <= sizeof(AppConfig.MySSID));
469.  memcpypgm2ram(AppConfig.MySSID, (ROM void*)MY_DEFAULT_SSID_NAME,
sizeof(MY_DEFAULT_SSID_NAME));
470.  AppConfig.SsidLength = sizeof(MY_DEFAULT_SSID_NAME) - 1;
471.  AppConfig.SecurityMode = MY_DEFAULT_WIFI_SECURITY_MODE;
472.  if (AppConfig.SecurityMode == WF_SECURITY_WEP_40)
473.  {
474.  AppConfig.WepKeyIndex = MY_DEFAULT_WEP_KEY_INDEX;
475.  memcpypgm2ram(AppConfig.SecurityKey, (ROM void*)MY_DEFAULT_WEP_KEYS_40,
sizeof(MY_DEFAULT_WEP_KEYS_40) - 1);
476.  AppConfig.SecurityKeyLength = sizeof(MY_DEFAULT_WEP_KEYS_40) - 1;
477.  }
478.  else if (AppConfig.SecurityMode == WF_SECURITY_WEP_104)
479.  {
480.  AppConfig.WepKeyIndex = MY_DEFAULT_WEP_KEY_INDEX;
481.  memcpypgm2ram(AppConfig.SecurityKey, (ROM void*)MY_DEFAULT_WEP_KEYS_104,
sizeof(MY_DEFAULT_WEP_KEYS_104) - 1);
482.  AppConfig.SecurityKeyLength = sizeof(MY_DEFAULT_WEP_KEYS_104) - 1;
483.  }
484.  AppConfig.networkType = MY_DEFAULT_NETWORK_TYPE;
485.  AppConfig.dataValid = 0;
486.  #endif
487.
488.  // Compute the checksum of the AppConfig defaults as loaded from ROM
489.  wOriginalAppConfigChecksum = CalcIPChecksum((BYTE*)&AppConfig, sizeof(AppConfig));
490.
491.  #if defined(EEPROM_CS_TRIS)
492.  NVM_VALIDATION_STRUCT NVMValidationStruct;
493.
494.  // Check to see if we have a flag set indicating that we need to
495.  // save the ROM default AppConfig values.
496.  if(vNeedToSaveDefaults)
497.  SaveAppConfig(&AppConfig);
498.
499.  // Read the NVMValidation record and AppConfig struct out of EEPROM/Flash
500.  XEEReadArray(0x0000, (BYTE*)&NVMValidationStruct, sizeof(NVMValidationStruct));
501.  XEEReadArray(sizeof(NVMValidationStruct), (BYTE*)&AppConfig, sizeof(AppConfig));
502.
503.  // Check EEPROM/Flash validitity. If it isn't valid, set a flag so
504.  // that we will save the ROM default values on the next loop
505.  // iteration.
506.  if((NVMValidationStruct.wConfigurationLength != sizeof(AppConfig)) ||
507.  (NVMValidationStruct.wOriginalChecksum != wOriginalAppConfigChecksum) ||
508.  (NVMValidationStruct.wCurrentChecksum != CalcIPChecksum((BYTE*)&AppConfig,
sizeof(AppConfig))))
```

```
509. {
510. // Check to ensure that the vNeedToSaveDefaults flag is zero,
511. // indicating that this is the first iteration through the do
512. // loop. If we have already saved the defaults once and the
513. // EEPROM/Flash still doesn't pass the validity check, then it
514. // means we aren't successfully reading or writing to the
515. // EEPROM/Flash. This means you have a hardware error and/or
516. // SPI configuration error.
517. if(vNeedToSaveDefaults)
518. {
519. while(1);
520. }
521.
522. // Set flag and restart loop to load ROM defaults and save them
523. vNeedToSaveDefaults = 1;
524. continue;
525. }
526.
527. // If we get down here, it means the EEPROM/Flash has valid contents
528. // and either matches the ROM defaults or previously matched and
529. // was run-time reconfigured by the user. In this case, we shall
530. // use the contents loaded from EEPROM/Flash.
531. break;
532. #endif
533. break;
534.
535. }
536.
537.
538. #if defined (EZ_CONFIG_STORE)
539. // Set configuration for ZG from NVM
540. /* Set security type and key if necessary, convert from app storage to ZG driver
*/
541.
542. if (AppConfig.dataValid)
543. CFGCXT.isWifiDoneConfigure = 1;
544.
545. AppConfig.saveSecurityInfo = FALSE;
546. #endif // EZ_CONFIG_STORE
547.
548. }
549.
550. /*****************************************************************************
551. Function:
552. void SelfTest()
553.
554. Description:
555. This routine performs a self test of the hardware.
556.
557. Precondition:
558. None
559.
560. Parameters:
561. None - None
562.
563. Returns:
```

```
564. None
565.
566. Remarks:
567. None
568. ********************************************************************/
569. static void SelfTest(void)
570. {
571. char value = 0;
572. char* buf[32];
573.
574. // Configure Sensor Serial Port
575. UARTConfigure(SENSOR_UART, UART_ENABLE_PINS_TX_RX_ONLY);
576. UARTSetFifoMode(SENSOR_UART, UART_INTERRUPT_ON_TX_NOT_FULL |
UART_INTERRUPT_ON_RX_NOT_EMPTY);
577. UARTSetLineControl(SENSOR_UART, UART_DATA_SIZE_8_BITS | UART_PARITY_NONE |
UART_STOP_BITS_1);
578. UARTSetDataRate(SENSOR_UART, GetPeripheralClock(), 9600);
579. UARTEnable(SENSOR_UART, UART_ENABLE_FLAGS(UART_PERIPHERAL | UART_RX | UART_TX));
580.
581. // Verify MRF24WB/G0MA MAC Address
582. if(AppConfig.MyMACAddr.v[0] == 0x00 && AppConfig.MyMACAddr.v[1] == 0x1E)
583. {
584. //*****************************************************************
585. // Prints a label using ESC/P commands to a Brother PT-9800PCN printer
586. //*****************************************************************
587. // Send ESC/P Commands to setup printer
588. UARTTxBuffer("\033ia\000\033@\033X\002",9); // ESC i a 0 = Put Printer in ESC/P
Mode
589. // ESC @ = Reset Printer to Default settings
590. // ESC X 2 = Specify Character Size
591. // Send the Info to Print for the MAC Address label
592. UARTTxBuffer("MRF24WB0MA\r",11);
593. sprintf((char *)buf,"MAC:
%02X%02X%02X%02X%02X%02X",AppConfig.MyMACAddr.v[0],AppConfig.MyMACAddr.v[1],AppConfig.M
yMACAddr.v[2]
594. UARTTxBuffer((char *)buf, strlen((const char *)buf));
595.
596. // Print the label
597. UARTTxBuffer("\f",1);
598.
599. // Toggle LED's
600. while(1)
601. {
602. LED0_IO = value;
603. LED1_IO = value >> 1;
604. LED2_IO = value >> 2;
605.
606. DelayMs(400);
607.
608. if(value == 8)
609. value = 0;
610. else
611. value++;
612.
613. }
614. }
```

```
615. else // MRF24WG0MA Failure
616. {
617. while(1)
618. {
619. LEDS_ON();
620. DelayMs(700);
621. LEDS_OFF();
622. DelayMs(700);
623. }
624. }
625. }
626.
627. /*****************************************************************************
628. Function:
629. void UARTTxBuffer(char *buffer, UINT32 size)
630.
631. Description:
632. This routine sends data out the Sensor UART port.
633.
634. Precondition:
635. None
636.
637. Parameters:
638. None - None
639.
640. Returns:
641. None
642.
643. Remarks:
644. None
645. *****************************************************************************/
646. void UARTTxBuffer(char *buffer, UINT32 size)
647. {
648. while(size)
649. {
650. while(!UARTTransmitterIsReady(SENSOR_UART))
651. ;
652.
653. UARTSendDataByte(SENSOR_UART, *buffer);
654.
655. buffer++;
656. size--;
657. }
658.
659. while(!UARTTransmissionHasCompleted(SENSOR_UART));
660. }
661.
662. /*****************************************************************************
663. Function:
664. void DisplayIPValue(IP_ADDR IPVal)
665.
666. Description:
667. This routine formats and prints the current IP Address.
668.
669. Precondition:
670. None
```

```
671.
672. Parameters:
673. None - None
674.
675. Returns:
676. None
677.
678. Remarks:
679. None
680. ********************************************************************/
681. void DisplayIPValue(IP_ADDR IPVal)
682. {
683. printf("%u.%u.%u.%u", IPVal.v[0], IPVal.v[1], IPVal.v[2], IPVal.v[3]);
684. }
685.
686. /********************************************************************
687. Function:
688. void SaveAppConfig(const APP_CONFIG *ptrAppConfig)
689.
690. Description:
691. This routine saves the AppConfig into EEPROM.
692.
693. Precondition:
694. None
695.
696. Parameters:
697. None - None
698.
699. Returns:
700. None
701.
702. Remarks:
703. None
704. ********************************************************************/
705. void SaveAppConfig(const APP_CONFIG *ptrAppConfig)
706. {
707. NVM_VALIDATION_STRUCT NVMValidationStruct;
708.
709. // Get proper values for the validation structure indicating that we can use
710. // these EEPROM/Flash contents on future boot ups
711. NVMValidationStruct.wOriginalChecksum = wOriginalAppConfigChecksum;
712. NVMValidationStruct.wCurrentChecksum = CalcIPChecksum((BYTE*)ptrAppConfig,
sizeof(APP_CONFIG));
713. NVMValidationStruct.wConfigurationLength = sizeof(APP_CONFIG);
714.
715. #if 0
716. // Write the validation struct and current AppConfig contents to EEPROM/Flash
717. XEEBeginWrite(0x0000);
718. XEEWriteArray((BYTE*)&NVMValidationStruct, sizeof(NVMValidationStruct));
719. XEEWriteArray((BYTE*)ptrAppConfig, sizeof(APP_CONFIG));
720. #endif
721. }
722.
723. #if defined (EZ_CONFIG_STORE)
724. /********************************************************************
725. Function:
```

```
726. void RestoreWifiConfig(void)
727.
728. Description:
729. This routine performs reboot when SW0 is pressed.
730.
731. Precondition:
732. None
733.
734. Parameters:
735. None - None
736.
737. Returns:
738. None
739.
740. Remarks:
741. None
742. *****************************************************************************/
743. void RestoreWifiConfig(void)
744. {
745. #if 0
746. XEEBeginWrite(0x0000);
747. XEEWrite(0xFF);
748. XEEWrite(0xFF);
749. XEEEndWrite();
750. #endif
751. // reboot here...
752. //LED_PUT(0x00);
753. while(SW0_IO == 0u);
754. Reset();
755. }
756. #endif // EZ_CONFIG_STORE
757.
```

HWP MRF24W XC32.h

```
1. /*******************************************************************
2. *
3. * Hardware specific definitions for:
4. * - WiFi G Demo Board
5. * ~ PIC32MX695F512H
6. * ~ MRF24WG0MA
7. *
8. ********************************************************************
9. * FileName: HWP MRF24W XC32.h
10. * Dependencies: Compiler.h
11. * Processor: PIC32MX695F512H
12. * Compiler: Microchip XC32 Compiler
13. * Company: Microchip Technology, Inc.
14. *
15. * Software License Agreement
16. *
17. * Copyright (C) 2002-2013 Microchip Technology Inc. All rights
18. * reserved.
```

```
19. *
20. * Microchip licenses to you the right to use, modify, copy, and
21. * distribute:
22. * (i) the Software when embedded on a Microchip microcontroller or
23. * digital signal controller product ("Device") which is
24. * integrated into Licensee's product; or
25. * (ii) ONLY the Software driver source files ENC28J60.c, ENC28J60.h,
26. * ENCX24J600.c and ENCX24J600.h ported to a non-Microchip device
27. * used in conjunction with a Microchip ethernet controller for
28. * the sole purpose of interfacing with the ethernet controller.
29. *
30. * You should refer to the license agreement accompanying this
31. * Software for additional information regarding your rights and
32. * obligations.
33. *
34. * THE SOFTWARE AND DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT
35. * WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT
36. * LIMITATION, ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR A
37. * PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL
38. * MICROCHIP BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR
39. * CONSEQUENTIAL DAMAGES, LOST PROFITS OR LOST DATA, COST OF
40. * PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
41. * BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE
42. * THEREOF), ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER
43. * SIMILAR COSTS, WHETHER ASSERTED ON THE BASIS OF CONTRACT, TORT
44. * (INCLUDING NEGLIGENCE), BREACH OF WARRANTY, OR OTHERWISE.
45. *
46. *
47. * Author Date Comment
48. *~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
49. * Amy Ong 10/22/2012 Created for WiFi G Demo Board to conform to existing MLA file
structures
50. ********************************************************************/
51.
52. #ifndef HARDWARE_PROFILE_H
53. #define HARDWARE_PROFILE_H
54.
55. // Set configuration fuses (but only in MainDemo.c where THIS_IS_STACK_APPLICATION
is defined)
56. #if defined(THIS_IS_STACK_APPLICATION)
57. // DEVCFG0.CP Code-protect bit
58. // DEVCFG1.FNOSC Oscillator selection bits
59. // DEVCFG1.FSOSCEN Secondary oscillator enable bit (Disabled)
60. // DEVCFG1.POSCMOD Primary oscillator configuration bits (Disabled)
61. // DEVCFG1.FPBDIV Peripheral bus clock divisor
62. // DEVCFG1.FWDTEN Watchdog timer enable bit
63. // DEVCFG2.FPLLIDIV PLL input divider bits
64. // DEVCFG2.FPLLMUL PLL multiplier bits
65. // DEVCFG2.FPLLODIV Default postscalar for PLL bits
66. #pragma config FNOSC = FRCPLL, FPLLIDIV = DIV_2, FPLLMUL = MUL_20, FPLLODIV =
DIV_2, FPBDIV = DIV_1, FWDTEN = OFF, POSCMOD = OFF, FSOSCEN 67. #endif
68.
69. #define GetSystemClock() (40000000ul) // 8MHz/FPLLIDIV*FPLLMUL/FPLLODIV
70. #define GetInstructionClock() (GetSystemClock()/1)
71. #define GetPeripheralClock() (GetInstructionClock()/1) // Set your divider
according to your Peripheral Bus Frequency configuration fuse 72.
```

```
73.
74. // ================================================
75. // Hardware mappings
76. // ================================================
77.
78.
79. //----------------------------
80. // LED and Button I/O pins
81. //----------------------------
82. #define LED0_TRIS (TRISEbits.TRISE0) // Ref D10 Green
83. #define LED0_IO (LATEbits.LATE0)
84. #define LED1_TRIS (TRISFbits.TRISF1) // Ref D9 Yellow
85. #define LED1_IO (LATFbits.LATF1)
86. #define LED2_TRIS (TRISFbits.TRISF0) // Ref D8 Red
87. #define LED2_IO (LATFbits.LATF0)
88.
89. #define LEDS_ON() {LED0_ON(); LED1_ON(); LED2_ON();}
90. #define LEDS_OFF() {LED0_OFF(); LED1_OFF(); LED2_OFF();}
91.
92. #define LED0_ON() LATESET = BIT_0;
93. #define LED0_OFF() LATECLR = BIT_0;
94. #define LED0_INV() LATEINV = BIT_0;
95.
96. #define LED1_ON() LATFSET = BIT_1;
97. #define LED1_OFF() LATFCLR = BIT_1;
98. #define LED1_INV() LATFINV = BIT_1;
99.
100. #define LED2_ON() LATFSET = BIT_0;
101. #define LED2_OFF() LATFCLR = BIT_0;
102. #define LED2_INV() LATFINV = BIT_0;
103.
104. #define SW0_TRIS (TRISDbits.TRISD9)
105. #define SW0_IO (PORTDbits.RD9)
106.
107. #define VBAT_TRIS (TRISBbits.TRISB0) // Battery level ADC input
108.
109. // Added to support EZ_CONFIG_STORE
110. // 25LC256 I/O pins
111. //#define EEPROM_CS_TRIS (TRISDbits.TRISD12)
112. #define EEPROM_CS_IO (LATDbits.LATD12)
113. #define EEPROM_SCK_TRIS (TRISGbits.TRISG6)
114. #define EEPROM_SDI_TRIS (TRISGbits.TRISG7)
115. #define EEPROM_SDO_TRIS (TRISGbits.TRISG8)
116. #define EEPROM_SPI_IF (IFS1bits.SPI2RXIF)
117. #define EEPROM_SSPBUF (SPI2BUF)
118. #define EEPROM_SPICON1 (SPI2CON)
119. #define EEPROM_SPICON1bits (SPI2CONbits)
120. #define EEPROM_SPIBRG (SPI2BRG)
121. #define EEPROM_SPISTAT (SPI2STAT)
122. #define EEPROM_SPISTATbits (SPI2STATbits)
123.
124.
125. //----------------------------
126. // MRF24WG0MA/B WiFi I/O pins
127. //----------------------------
128.
```

```
129. #define WF_CS_TRIS (TRISGbits.TRISG9)
130. #define WF_CS_IO (LATGbits.LATG9)
131. #define WF_SDI_TRIS (TRISGbits.TRISG7)
132. #define WF_SCK_TRIS (TRISGbits.TRISG6)
133. #define WF_SDO_TRIS (TRISGbits.TRISG8)
134. #define WF_RESET_TRIS (TRISDbits.TRISD1)
135. #define WF_RESET_IO (LATDbits.LATD1)
136.
137. // NOTE:
138. // Wifi comm demo declares WF_INT_VECTOR as _EXTERNAL_1_VECTOR and used in
WF_Eint.c.
139. // Wifi G demo adopts generic approach taken by demo/console/ezconfig by defining
MRF24W_IN_SPI1.
140. // _EXTERNAL_1_VECTOR = MRF24W_IN_SPI1
141. #define MRF24W_IN_SPI1
142.
143. #define WF_INT_TRIS (TRISDbits.TRISD8) // INT1
144. #define WF_INT_IO (PORTDbits.RD8)
145.
146. #define WF_HIBERNATE_TRIS (TRISEbits.TRISE4)
147. #define WF_HIBERNATE_IO (PORTEbits.RE4)
148.
149. #define WF_INT_EDGE (INTCONbits.INT1EP)
150. #define WF_INT_IE (IEC0bits.INT1IE)
151. #define WF_INT_IF (IFS0bits.INT1IF)
152. #define WF_INT_IE_CLEAR IEC0CLR
153. #define WF_INT_IF_CLEAR IFS0CLR
154. #define WF_INT_IE_SET IEC0SET
155. #define WF_INT_IF_SET IFS0SET
156. #define WF_INT_BIT 0x00000080
157. #define WF_INT_IPCSET IPC1SET
158. #define WF_INT_IPCCLR IPC1CLR
159. #define WF_INT_IPC_MASK 0xFF000000
160. #define WF_INT_IPC_VALUE 0x0C000000
161.
162. #define WF_SSPBUF (SPI2BUF)
163. #define WF_SPISTAT (SPI2STAT)
164. #define WF_SPISTATbits (SPI2STATbits)
165.
166. #define WF_SPICON1 (SPI2CON)
167. #define WF_SPICON1bits (SPI2CONbits)
168. #define WF_SPI_IE_CLEAR IEC1CLR
169. #define WF_SPI_IF_CLEAR IFS1CLR
170. #define WF_SPI_INT_BITS 0x000000e0
171. #define WF_SPI_BRG (SPI2BRG)
172. #define WF_MAX_SPI_FREQ (10000000ul) // Hz
173.
174. //----------------------------
175. // UART to Telnet Mapping
176. //----------------------------
177. #define BusyUART() (TelnetOutFree() ? 0 : (StackTask(), 1))
178. #define putcUART putchar
179. #define putrsUART(a) fputs((const char*)a,(FILE *)stdout)
180. #define putsUART(a) fputs((const char*)a,(FILE *)stdout)
181. #define DataRdyUART() TelnetInChars()
182. #define ReadUART() TelnetGet()
```

```
183.
184. //----------------------------
185. // Sensor Port Mapping
186. //----------------------------
187. #define SENSOR_UART UART2
188.
189. #endif // #ifndef HARDWARE_PROFILE_H
190.
```

Anexo 2

Ficheros *ModbusQuery.class*, *Config.class y Consumos.class* de la aplicación cliente ModbusTCP.

Config.class

```
1. public class Config {
2. private static Connection c = null;
3. private static Statement stmt = null;
4. private static final String[] modColumnName={"Módulo", "Modbus Address", "PLC"};
5. private static final String[] plcColumnName={"PLC", "IP"};
6. private static ModTable modTable;
7. private static PlcTable plcTable;
8. private static Config INSTANCE;
9. //Vector vector=new Vector();
10.
11. private Config(){
12.    getModTable();
13.    getPlcTable();
14.
15.
16.
17. }
18.
19. // creador sincronizado para protegerse de posibles problemas multi-hilo
20. // otra prueba para evitar instanciación múltiple
21. private synchronized static void createInstance() {
22.    if (INSTANCE == null) {
23.          INSTANCE = new Config();
24.    }
25. }
26.
27. public static Config getInstance() {
28.    if (INSTANCE == null) createInstance();
29. return INSTANCE;
30. }
31.
32.
33. public static JTable getModTable(){
34.
35.    String query="SELECT * FROM MODULOS";
36.    Vector<Vector<String>> rowData = new Vector<Vector<String>>();
37.    Vector<String> data;
38.    Vector<String> columnNames = new Vector<String>(Arrays.asList(modColumnName));
39.
40. try {
41.    Class.forName("org.sqlite.JDBC");
42.    c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
43.    c.setAutoCommit(false);
44.    System.out.println("Opened database successfully");
45.    stmt = c.createStatement();
46.    ResultSet rs=stmt.executeQuery(query);
47.    while ( rs.next() ) {
```

```
48.          data = new Vector<String>();
49.
50.          data.add(rs.getString("name"));
51.          data.add(rs.getString("address"));
52.          data.add(rs.getString("plc"));
53.          rowData.add(data);
54.
55.    }
56.    rs.close();
57.    stmt.close();
58.    c.close();
59.    } catch ( Exception e ) {
60.    System.err.println( e.getClass().getName() + ": " + e.getMessage() );
61.    System.exit(0);
62. }
63. System.out.println("ModTable OK");
64.
65.
66.
67. return modTable=new ModTable(rowData,columnNames);
68. }
69. public static JTable getPlcTable(){
70.    String query="SELECT * FROM PLCS";
71.    Vector<Vector<String>> rowData = new Vector<Vector<String>>();
72.    Vector<String> data;
73.    Vector<String> columnNames = new Vector<String>(Arrays.asList(plcColumnName));
74.
75. //Vector data=new Vector();
76.
77.    try {
78.          Class.forName("org.sqlite.JDBC");
79.          c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
80.          c.setAutoCommit(false);
81.          System.out.println("Opened database successfully");
82.          stmt = c.createStatement();
83.          ResultSet rs=stmt.executeQuery(query);
84.          while ( rs.next() ) {
85.          data = new Vector<String>();
86.
87.          data.add(rs.getString("name"));
88.          data.add(rs.getString("ip"));
89.          rowData.add(data);
90.
91.    }
92.    rs.close();
93.    stmt.close();
94.    c.close();
95.    } catch ( Exception e ) {
96.          System.err.println( e.getClass().getName() + ": "+e.getMessage());
97.          System.exit(0);
98.    }
99. System.out.println("PLCTAble OK");
100.
101.
102.
103. return plcTable=new PlcTable(rowData,columnNames);
```

```
104.
105. }
106.
107.
108. public static void addMod(String name,int address,String plc){
109. String query="INSERT INTO MODULOS (name,address,plc) " + "VALUES ('"+ name +"', '"
+ Integer.toString(address) + "', '" + plc +
110. System.out.println(query);
111.
112. try {
113.  Class.forName("org.sqlite.JDBC");
114.  c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
115.  c.setAutoCommit(false);
116.  System.out.println("Opened database successfully");
117.
118.  stmt = c.createStatement();
119.  String sql = query;
120.  stmt.executeUpdate(sql);
121.  stmt.close();
122.  c.commit();
123.  c.close();
124. } catch ( Exception e ) {
125.  System.err.println( e.getClass().getName() + ": " + e.getMessage() );
126.  System.exit(0);
127. }
128.  System.out.println("Records created successfully");
129. }
130.
131.
132. public static void addPlc(String name,String ip){
133. String query="INSERT INTO PLCS (NAME,IP) " + "VALUES ('" + name +"', '" + ip +
"');";
134.
135.
136. try {
137.  Class.forName("org.sqlite.JDBC");
138.  c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
139.  c.setAutoCommit(false);
140.  System.out.println("Opened database successfully");
141.
142.  stmt = c.createStatement();
143.  String sql = query;
144.  stmt.executeUpdate(sql);
145.  stmt.close();
146.  c.commit();
147.  c.close();
148. } catch ( Exception e ) {
149.  System.err.println( e.getClass().getName() + ": " + e.getMessage() );
150.  System.exit(0);
151. }
152. System.out.println("Records created successfully");
153. }
154.
155. public void removeMod(String name){
156. String query="DELETE FROM 'MODULOS' WHERE NAME='"+name+"';";
157. try {
```

```
158.   Class.forName("org.sqlite.JDBC");
159.    c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
160.   c.setAutoCommit(false);
161.   System.out.println("Opened database successfully");
162.
163.   stmt = c.createStatement();
164.   String sql = query;
165.   stmt.executeUpdate(sql);
166.   stmt.close();
167.   c.commit();
168.   c.close();
169. } catch ( Exception e ) {
170.   System.err.println( e.getClass().getName() + ": " + e.getMessage() );
171.   System.exit(0);
172. }
173.   System.out.println("Records deleted successfully");
174. }
175.
176. public void removePlc(String name){
177.   String query="DELETE FROM 'PLCS' WHERE NAME='"+name+"';";
178.   System.out.println(query);
179. try {
180.   Class.forName("org.sqlite.JDBC");
181.   c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
182.   c.setAutoCommit(false);
183.   System.out.println("Opened database successfully");
184.
185.   stmt = c.createStatement();
186.   String sql = query;
187.   stmt.executeUpdate(sql);
188.   stmt.close();
189.   c.commit();
190.   c.close();
191. } catch ( Exception e ) {
192.   System.err.println( e.getClass().getName() + ": " + e.getMessage() );
193.   System.exit(0);
194. }
195. System.out.println("Records deleted successfully");
196. }
197.
198.
199.
200.
201.
202. public void updateMod(String key,String name,String address,String plc){
203. String query="UPDATE 'MODULOS' SET
NAME='"+name+"',ADDRESS='"+address+"',PLC='"+plc+"' WHERE NAME='"+key+"';" ;
204. System.out.println(query);
205. try {
206.   Class.forName("org.sqlite.JDBC");
207.   c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
208.   c.setAutoCommit(false);
209.   System.out.println("Opened database successfully");
210.
211.   stmt = c.createStatement();
212.   String sql = query;
```

```
213.   stmt.executeUpdate(sql);
214.   stmt.close();
215.   c.commit();
216.   c.close();
217. } catch ( Exception e ) {
218.   System.err.println( e.getClass().getName() + ": " + e.getMessage() );
219.   System.exit(0);
220. }
221. System.out.println("Records deleted successfully");
222.
223.
224. }
225. public void updatePlc(String key,String name,String ip){
226. String query="UPDATE 'PLCS' SET NAME='"+name+"',IP='"+ip+"' WHERE NAME='"+key+"';"
;
227. System.out.println(query);
228. try {
229.   Class.forName("org.sqlite.JDBC");
230.   c = DriverManager.getConnection("jdbc:sqlite:appconfig.db");
231.   c.setAutoCommit(false);
232.   System.out.println("Opened database successfully");
233.
234.   stmt = c.createStatement();
235.   String sql = query;
236.   stmt.executeUpdate(sql);
237.   stmt.close();
238.   c.commit();
239.   c.close();
240. } catch ( Exception e ) {
241.   System.err.println( e.getClass().getName() + ": " + e.getMessage() );
242.   System.exit(0);
243. }
244. System.out.println("Records deleted successfully");
245.
246.
247. }
248.
249.
250.
251. public JTable getPlc(){
252.   return plcTable;
253.
254. }
255. public JTable getMod(){
256.   return modTable;
257.
258. }
259.
260.
261. }
262
```

ModbusQuery.class

```
1.
```

```
2. public class ModbusQuery {
3.
4. private TCPMasterConnection con = null; //the connection
5. private ModbusTCPTransaction trans = null; //the transaction
6. private ReadMultipleRegistersRequest req = null; //the request
7. private ReadMultipleRegistersResponse res = null; //the response
8. private InetAddress addr = null; //the slave's address
9. private int count; //the number of DI's to read
10. private int repeat;
11. private int port;
12. private int ref;
13. private static ModbusQuery INSTANCE=null;
14.
15.
16. private ModbusQuery(String addr,int port,int ref, int count,int repeat)
throws Exception{
17.
18.   this.addr= InetAddress.getByName(addr);
19.   this.count=count;//los registros que leemos son de 32 bits
20.   this.repeat=repeat;
21.   this.port=port;
22.   this.ref=ref;
23.   //Open the connection
24.   con = new TCPMasterConnection(this.addr);
25.   con.setPort(port);
26.   con.connect();
27.   //Prepare the request
28.   req = new ReadMultipleRegistersRequest(ref,1);
29.   req.setUnitID(1);
30.   //Prepare the transaction
31.   trans = new ModbusTCPTransaction(con);
32.   trans.setRequest(req);
33.
34.
35. }
36. public int ExecuteSingleQuery() throws Exception{
37.
38.   int k = 0;
39.   int result;
40.
41.   //Prepare the request
42.   req = new ReadMultipleRegistersRequest(this.ref,1);
43.   req.setUnitID(1);
44.   //Prepare the transaction
45.   trans = new ModbusTCPTransaction(con);
46.   trans.setRequest(req);
47.   trans.execute();
48.   res = (ReadMultipleRegistersResponse) trans.getResponse();
49.   result=(int)res.getRegisterValue(0);
50.   return result;
51.
```

```
52.
53.
54. }
55.
56. public TCPMasterConnection getCon() {
57.   return con;
58. }
59.
60.
61. public void setCon(TCPMasterConnection con) {
62.   this.con = con;
63. }
64.
65.
66. public ModbusTCPTransaction getTrans() {
67.   return trans;
68. }
69.
70.
71. public void setTrans(ModbusTCPTransaction trans) {
72.   this.trans = trans;
73. }
74.
75.
76. public ReadMultipleRegistersRequest getReq() {
77.   return req;
78. }
79.
80.
81. public void setReq(ReadMultipleRegistersRequest req) {
82.   this.req = req;
83. }
84.
85.
86. public ReadMultipleRegistersResponse getRes() {
87.   return res;
88. }
89.
90.
91. public void setRes(ReadMultipleRegistersResponse res) {
92.   this.res = res;
93. }
94.
95.
96. public InetAddress getAddr() {
97.   return addr;
98. }
99.
100.
101. public void setAddr(InetAddress addr) {
102.   this.addr = addr;
```

```
103. }
104.
105.
106. public int getCount() {
107.  return count;
108. }
109.
110.
111. public void setCount(int count) {
112.  this.count = count;
113. }
114.
115.
116. public int getRepeat() {
117.  return repeat;
118. }
119.
120.
121. public void setRepeat(int repeat) {
122.  this.repeat = repeat;
123. }
124.
125.
126. public int getPort() {
127.  return port;
128. }
129.
130.
131. public void setPort(int port) {
132.  this.port = port;
133. }
134.
135.
136. public int getRef() {
137.  return ref;
138. }
139.
140.
141. public void setRef(int ref) {
142.  this.ref = ref;
143. }
144.
145.
146. private synchronized static void createInstance(String addr,int port,int
ref, int count,int repeat) throws Exception {
147.  if (INSTANCE == null) {
148.        INSTANCE = new ModbusQuery(addr,port,ref,count,repeat);
149.  }
150. }
151.
```

```
152. public static ModbusQuery getInstance(String addr,int port,int ref, int
count,int repeat) throws Exception {
153.  if (INSTANCE == null) createInstance(addr,port,ref,count,repeat);
154.        return INSTANCE;
155.  }
156. public void close(){
157.  con.close();
158. }
159. }
160.
161.
```

Consumo.class

```
1. public class Consumos extends JPanel implements Runnable{
2. /**
3. *
4. */
5.    private static final long serialVersionUID = 6867570462477882608L;
6.    private final static String [] columnNames={"Módulo","Consumo"};
7.    String [][] array=new String[3][3];
8.    private JTable table;
9.    private JScrollPane scrollPanel;
10.   TableModel tableModel;
11.   Thread thread;
12.   boolean stop;
13.   public Consumos() {
14.         this.setBackground(Color.WHITE);
15.         try {
16.               tableModel=new TableModel(null,columnNames);
17.         } catch (Exception e) {
18.
19.               e.printStackTrace();
20.         }
21.         table =new ResultTable(tableModel);
22.         scrollPanel=new JScrollPane(table);
23.         scrollPanel.setBounds(0, 0, 654, 455);
24.         table.setFillsViewportHeight(true);
25.         setLayout(null);
26.         this.add(scrollPanel);
27.         this.setVisible(true);
28. }
29.
30.
31. @Override
32. public void run() {
33.   int i=0;
34.   while(!stop){
35.         i++;
36.
37.         System.out.println("\nIteracion:"+i);
```

```
38.          tableModel=null;
39.          table=null;
40.          this.remove(scrollPanel);
41.          try {
42.                tableModel=new
                   TableModel(ResultTable.getRegisters(),columnNames);
43.                table =new ResultTable(tableModel);
44.                scrollPanel=new JScrollPane(table);
45.                scrollPanel.setBounds(0, 0, 654, 455);
46.                table.setFillsViewportHeight(true);
47.                setLayout(null);
48.                this.add(scrollPanel);
49.                this.setVisible(true);
50.
51.
52.                Thread.sleep(100);
53.
54.                }catch (Exception e) {e.printStackTrace(); }
55.
56.    }
57.    try {
58.          ModbusQuery.getInstance(null, 0, 0, 0, 0).close();
59.    } catch (Exception e) {
60.
61.          e.printStackTrace();
62.    }
63.
64. }
65.    public void start(){
66.          thread =new Thread(this);
67.          stop=false;
68.          this.thread.start();
69.    }
70.
71.    public void stop(){
72.
73.          stop=true;
74.          thread=null;
75.    }
76.    public JTable getTable(){
77.
78.          return table;
79.
80.    }
81.
82. }
83.
84.
85.
```