UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática

Universidad Politécnica de Valencia

Internet de las Cosas: extensión del protocolo UDP sobre Arduino

Proyecto Final de Carrera

Ingeniería Técnica en Informática de Sistemas

Autor: Gastón Alejandro Zunino Luna

Director: José Vicente Busquets Mataix

Julio 2014

Dedicado:

A mis padres, Héctor y Ester, por su apoyo y su amor;

A mi hermana Aixa, por su cariño y compresión;

A mis abuelos, por esperar este momento;

A Bruno y Nero, por iluminar mis días.

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mis padres por su apoyo incondicional durante todos estos años y por haberme acompañado, aconsejado y animado en los momentos más difíciles de la realización de este proyecto.

En segundo lugar, agradezco profundamente la colaboración inestimable de mi hermana, por su amistad, por su comprensión y buena disposición.

También quiero expresar mi más sincero agradecimiento al profesor José Vicente Busquets Mataix por su valiosa guía, consejos e infinita paciencia.

Esperando no olvidarme de nadie, agradezco a toda persona que directa o indirectamente haya colaborado para que este trabajo hoy sea una realidad. Gracias a todos.

RESUMEN

El presente proyecto tiene como principal objetivo la utilización de hardware libre basado en la plataforma Arduino para el desarrollo de aplicaciones para Internet de las Cosas.

Se presentará una alternativa a las soluciones actuales de comunicación entre dispositivos microcontroladores, que se apoya fundamentalmente en el uso del protocolo UDP al que se le ha añadido la característica de fiabilidad, aprovechando la posibilidad de integración con un sistema operativo de tiempo real y extendiendo otras funcionalidades como la actualización automática de firmware presente en el microcontrolador gracias al uso del protocolo TFTP.

Para hacer posible todo ello, se presentarán diversas herramientas software y configuraciones hardware para el desarrollo de aplicaciones que puedan comunicarse entre sí, opcionalmente mediante una arquitectura cliente/servidor, con el objetivo de crear una red de objetos inteligentes interconectados a través de Internet.

PALABRAS CLAVE: Internet de las Cosas, Arduino, Microcontrolador, ENC28J60, ChibiOS, UDP, Fiabilidad UDP, Bootloader, Firmware, TFTP

ÍNDICE

1. Introducción

1.1. Motivación

Internet de las Cosas (The Internet of Things o IoT) se refiere a la interacción entre objetos que puedan ser identificables de manera unívoca como también a sus representaciones virtuales mediante un tipo de estructura de red global con las características propias de Internet.

Los objetos serían capaces de comunicarse entre sí y de esta forma podrían tomar decisiones con la menor interacción humana posible.

Para que objetos de diferente naturaleza puedan coexistir y comunicarse entre sí, es necesario algún protocolo de comunicación mediante el cual los objetos puedan describirse a sí mismos y publicar a través de la red qué son capaces de hacer, cómo reaccionar ante un evento externo y de qué forma interactuar en consecuencia con el resto de objetos.

Con los progresivos avances en miniaturización de microcontroladores, bastará con integrar un chip de dimensiones muy reducidas en cualquier objeto del trabajo, hogar o ciudad.

Se abre por tanto, un extenso abanico de posibilidades y aplicaciones en Internet de las Cosas: recolección en tiempo real de datos y cómputo de estadísticas para mejorar el aparcamiento y flujo de tráfico (destinado a reducir la polución y accidentes), automatización de cálculo de rutas óptimas en automóviles como la interconexión entre los mismos, gestión de cualquier tipo de objeto del hogar desde luces, cafeteras o puertas de garajes, calefacción, gestión de consumo eléctrico y ahorro energético, avisos de emergencia, compras inteligentes, entre otras.

Con la futura proliferación de redes basadas en el protocolo IPv6 (que extiende el rango de IPs del actual IPv4 hasta 2^{128} IPs diferentes), se estima que Internet de las Cosas pueda llegar a estar compuesto por miles de millones de dispositivos interconectados en los próximos años, pudiendo cada uno de los mismos obtener una IP única que les sirva como identificación e interconexión a Internet.

1.2. Objetivos

El objetivo principal del presente proyecto es utilizar una placa microcontroladora Arduino (basada en hardware libre) y explotar sus capacidades hardware para poder obtener datos desde de algún sensor y enviarlos a través de la red (Internet) como también recibir órdenes para activar algún actuador.

Se presentará también un servidor (en sus versiones para Arduino como en plataforma PC) que pueda aprovechar las características de Arduino de bajo consumo energético y

que sirva de enlace entre diferentes clientes, comunicándose entre sí a través de un identificador único como método de direccionamiento y que permita la consulta de las últimas muestras de datos recibidas desde cada cliente, como también la posibilidad de enviar órdenes a los mismos.

La idea básica es la de dotar a cada objeto de una serie de sensores conectados a un microcontrolador que pueda procesar en tiempo real los datos procedentes de dichos sensores que interpretan el mundo físico y así transmitirlos constantemente a través de la red para que puedan ser almacenados o procesados inmediatamente por cualquier otra entidad conectada a la red, como puede ser otro objeto cliente o servidor.

Se pretende añadir al hardware Arduino algunas funcionalidades adicionales que de manera predeterminada no están disponibles en dicha plataforma. Esto es posible gracias a la versión para Arduino del sistema operativo de tiempo real ChibiOS/RT, utilizada en el presente proyecto para cumplir con el objetivo de crear diferentes tareas mediante hilos de ejecución y usar elementos de sincronización entre los mismos.

Así también, la comunicación a través de la red (Internet) será posible mediante el módulo de red Ethernet ENC28J60 que ha ganado aceptación como solución económica frente al Ethernet Shield W5100.

Dado que dicho módulo dispone de una pila TCP/IP básica implementada por software y debido a las restricciones de cantidad de memoria en microcontroladores, se propone el diseño de la clase denominada *Socket*, que contempla solamente el envío y recepción de mensajes sin conexión basado en el protocolo UDP, suponiendo ésta elección un ahorro de memoria por conexión establecida y por tanto, haciéndolo más apropiado en el desarrollo de aplicaciones para Internet de las Cosas.

Por otra parte, se pretende facilitar el desarrollo de aplicaciones sobre microcontroladores para Internet de las Cosas que requieran el uso de llamadas al sistema pertenecientes a la API de sockets BSD. El establecimiento de conexiones de red se facilitará a través del uso de la clase Socket diseñada en C++ .

Para finalizar, dado que la principal motivación es que los objetos basados en microcontroladores puedan comunicarse a través de Internet, la actualización remota del firmware tomará una mayor importancia, por lo que se ha añadido esta característica, que de forma predeterminada no está disponible en dichos dispositivos.

1.3. Planificación

Para alcanzar los objetivos propuestos en el presente proyecto, se plantean las siguientes tareas:

- Presentar las ventajas que aportan otras alternativas de herramientas de desarrollo de software para la plataforma Arduino.

- Dotar al hardware Arduino de algunas funcionalidades adicionales que por defecto no están disponibles en dicha plataforma a través del sistema operativo de tiempo real ChibiOS/RT.

- Desarrollar un protocolo de comunicación de red básico apto para microcontroladores, teniendo en cuenta sus restricciones de memoria y capacidad de procesamiento y que permita la interconexión e intercambio de mensajes entre los clientes a través de un servidor (Arduino o PC).

- Desarrollar un servidor (en sus versiones para Arduino como en plataforma PC) que sirva de enlace entre diferentes clientes y que permita la consulta de las últimas muestras de datos recibidas desde cada cliente, como también la posibilidad de enviar órdenes a éstos.

- Modificar el código de inicio (gestor de arranque o *bootloader*) del microcontrolador para dotarlo de la posibilidad de descarga y actualización remota del firmware desde un servidor TFTP.

2. Descripción de componentes Hardware

2.1. Arduino Mega 2560

Características generales

El Arduino Mega 2560 es una placa microcontroladora basada en el microprocesador Atmega2560 del fabricante Atmel, perteneciente a la familia de arquitectura RISC conocida como AVR, de bajo consumo energético y de 8 bits.

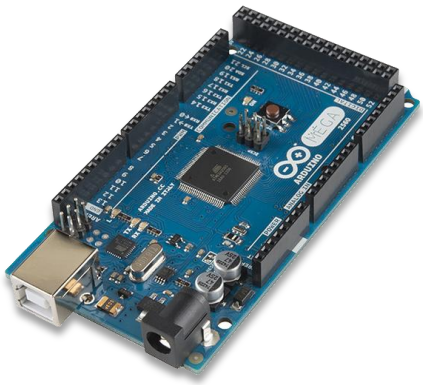

Figura 2.1. Arduino Mega 2560

Este modelo de Arduino dispone de las siguientes características:

Característica	Descripción
Microcontrolador	Atmega2560
Voltaje operativo	5V
Voltaje de entrada (recomendado)	7-12V
Voltaje de entrada (límite)	6-20V
Pines digitales de E/S	54 (de los cuales 15 proporcionan salida PWM)
Pines de entrada analógica	16
Corriente continua para pines E/S	40 mA
Corriente continua para pines de 3.3V	50 mA
Memoria Flash	256 KB de los cuales 8 KB son usados por el bootloader
SRAM	8 KB
EEPROM	4 KB
Velocidad de Reloj	16 MHz

Alimentación

El Arduino Mega 2560 dispone de dos fuentes de alimentación diferentes: mediante la conexión por el puerto USB o bien con una fuente de alimentación externa, de las cuales una es seleccionada automáticamente.

La alimentación externa puede ser desde un transformador de AC a DC o una batería, cuyo rango de voltajes de entrada va desde 6V hasta 20V, aunque con menos de 7V es posible que no sea suficiente para que el pin de 5V pueda abastecer el voltaje requerido para que la placa se mantenga en funcionamiento estable.

Por lo tanto, el rango recomendado es desde 7V hasta 12V. Los pines de alimentación se describen a continuación:

Nombre de pin	Descripción
VIN	Sirve para suministrar alimentación desde una fuente externa
5V	Se pueden obtener 5V pasando por un regulador
3V3	Se pueden obtener 3.3V con una corriente máxima de 50 mA
GND	Toma de masa del circuito o tensión de referencia de 0V
IOREF	Proporciona la referencia de tensión con la que opera el microcontrolador para cualquier *shield* o elemento acoplable

Entrada y Salida

El Arduino Mega 2560 proporciona 54 pines digitales que pueden ser usados como entrada o salida, usando las funciones *pinMode()*, *digitalWrite()* y *digitalRead()*. Funcionan a 5V y pueden proporcionar o recibir un máximo de 40 mA.

Cada pin tiene una resistencia de pull-up interna de 20-50 kOhmios. Además algunos pines están reservados para alguna función en particular:

- Serie: 0 (RX) y 1 (TX). Se utilizan para recibir (RX) y para transmitir (TX) datos serie y están directamente conectados a los pines serie del microcontrolador. A través de estos pines se puede conectar otra placa.

- Interrupciones externas: Son los pines 2 (interrupción 0), 3 (interrupción 1), 18 (interrupción 5), 19 (interrupción 4), 20 (interrupción 3) y 21 (interrupción 2). Estos pines pueden ser configurados para disparar una interrupción a nivel bajo (LOW) o alto (HIGH), flanco de bajada (FALLING) o de subida (RISING) o bien por cambio de nivel de alto a bajo o viceversa (CHANGE). Para establecer una rutina de servicio de interrupción (ISR) se utiliza la función *attachInterrupt()*.

- PWM: Pines del 2 al 13 y del 44 al 46. Proporcionan salida PWM de 8 bits con la función *analogWrite()*.

- SPI: Pines 50 (MISO), 51 (MOSI), 52 (SCK) y 53 (SS), son los pines utilizados para la comunicación SPI.

- LED: Existe un LED integrado en la placa conectado al pin 13. Cuando el pin está a nivel alto el LED está encendido y con nivel bajo permanece apagado.

- TWI: Pines 20 (SDA) y 21 (SCL) que soportan comunicación TWI o IC2.

Además de los pines digitales descritos anteriormente, también se dispone de 16 entradas analógicas, cada una de las cuales proporcionan 10 bits de resolución (1024 valores diferentes).

Cabe destacar el uso de los siguientes pines con función analógica:

- AREF: Voltaje de referencia para entradas analógicas. Usado mediante la función *analogReference()*.

- Reset: Poniendo esta línea a nivel bajo se puede reiniciar el microcontrolador. Se usa típicamente para agregar un botón de reset, pero inutilizaría al principal que viene integrado en la placa.

Comunicación

El Arduino Mega 2560 puede comunicarse con un PC, otro Arduino u otro microcontrolador. Dispone de comunicación serie y USB. Éste último método de comunicación se provee a modo de puerto COM virtual emulado por software en máquinas Windows o Linux y es detectado automáticamente.

La placa puede alimentarse desde el puerto de comunicaciones USB con una tensión de 5V.

Por otra parte también se dispone de comunicación TWI y SPI.

Programación

Para programar la placa Arduino Mega 2560, está disponible el software propio de Arduino aunque también es posible en el caso de Windows, la utilización de Visual Studio (requiere la instalación de algunos *plugins* adicionales).

La placa Arduino tiene un gestor de arranque o bootloader que permite subir a través del puerto USB, un nuevo código sin la necesidad de utilizar un programador externo.

Sin embargo, la programación del bootloader sí requiere de un programador externo con cabezal ICSP (In Circuit Serial Programming).

2.2. Elementos acoplables a Arduino

Puesto que Internet de las Cosas estaría compuesto por objetos capaces de interaccionar entre sí, debe existir algún mecanismo mediante el cual los objetos puedan percibir el mundo que los rodea.

Aquellos elementos que dan la capacidad a los objetos de obtener datos desde el exterior son los sensores.

Los objetos tendrían la capacidad de ejecutar órdenes como resultado de la interacción con otros objetos o de la intervención humana. Éstas órdenes se manifestarían en forma de representación física, como por ejemplo visualmente encendiendo una señal luminosa, de forma sonora como alarma, activando algún sistema mecánico, etc.

Por lo tanto, aquellos elementos que permitan algún cambio en el contexto del mundo real son los actuadores.

Los elementos acoplables a Arduino se conectan directamente a través de las diversas conexiones que proporcionan los pines de la placa (digitales o analógicas).

Para el caso de aquellos elementos externos digitales, se trabajará principalmente en base a dos valores, nivel bajo o nivel alto; en el caso de los analógicos, se permite cierta flexibilidad en comparación con los digitales, lo que implica que sean levemente más complicados de utilizar, ya que funcionan mediante la lectura/escritura de un voltaje de 0 a 5V, representables con 10 bits para la lectura, o con 8 bits para escritura, lo que proporciona una cantidad total de 1024 valores diferentes representables para la lectura y 256 valores representables para la escritura.

La mayoría de los dispositivos acoplables requieren librerías adicionales ya que se tratan de piezas hardware con funcionalidades específicas que no incorpora el hardware principal Arduino.

2.2.1. Sensores

Un sensor es un dispositivo que responde a una cantidad de entrada mediante la generación de una salida funcionalmente relacionada, por lo general en la forma de una señal eléctrica u óptica. Se trata de un dispositivo que mide y convierte una cantidad física en una señal que puede ser leída por un observador, normalmente un instrumento electrónico.

Pueden detectar magnitudes físicas o químicas, conocidas como variables de instrumentación y transformarlas en variables eléctricas.

Algunos ejemplos de variables de instrumentación pueden ser: temperatura, intensidad lumínica, distancia, aceleración, inclinación, presión, desplazamiento, fuerza, torsión, humedad, movimiento, pH, etc.

Como ejemplos de magnitud eléctrica existen: resistencia eléctrica, capacidad eléctrica, tensión eléctrica, corriente eléctrica, etc.

Aquellos sensores que estén permanentemente activados podrán tomar muestras del contexto que les rodea continuamente, transmitiendo dichas muestras a la placa Arduino al que estén conectados, siendo responsable éste último de interpretarlas, convertirlas a un formato adecuado si procede y decidir cómo actuar en consecuencia.

Los sensores digitales conectados a la placa Arduino deben ser inicializados como pin de salida con la función *pinMode()*. La lectura de datos se realiza mediante la llamada a la función *digitalRead()*.

Por otro lado, los sensores analógicos no requieren ninguna inicialización y para leer datos se utiliza la llamada a función *analogRead()*.

A continuación se citarán algunos ejemplos de sensores susceptibles de ser utilizados en dispositivos incorporados a objetos y que puedan tomar muestras del entorno en el que se encuentren:

Sensor de temperatura y humedad: éste sensor tiene la capacidad de obtener muestras de temperatura y humedad, ambos desde el mismo módulo, pudiéndose representar digitalmente la humedad ambiental y la temperatura. Un ejemplo es el modelo DHT11.

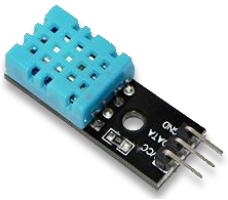

Figura 2.2. Sensor de temperatura y humedad DHT11

Sensor de luz: detecta el nivel de intensidad de la luz ambiental analógicamente.

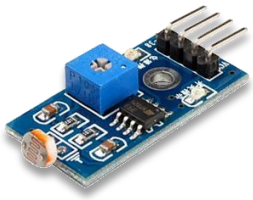

Figura 2.3. Sensor de luz

Sensor PIR: se trata de un elemento utilizado principalmente en el campo de la seguridad ya que es capaz de detectar el movimiento.

Figura 2.4. Sensor de detección de movimiento

Sensor de inclinación: su funcionamiento es equivalente al de un interruptor, que se utiliza como entrada digital. Cuando se encuentra nivelado, el interruptor queda abierto y cuando se inclina se cierra el interruptor que emite una señal en uno de sus pines.

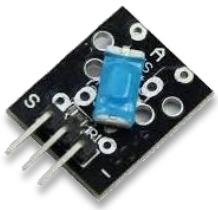

Figura 2.5. Sensor de inclinación

Sensor de ultrasonidos: se utiliza para medir a qué distancia se encuentra el obstáculo más cercano. El rango de distancia efectiva de este sensor varía entre 2cm y 500cm, con una resolución de 1 cm.

Figura 2.6. Sensor de ultrasonidos

Sensor acelerómetro y giroscopio: puede detectar la aceleración y rotaciones y dispone de una salida de 16 bits.

Figura 2.7. Sensor de aceleración y rotación

2.2.2. Actuadores

Un actuador es un dispositivo capaz de transformar la energía en movimiento, el cual puede ser utilizado para aplicar una fuerza.

Típicamente se trata de un dispositivo mecánico que toma la energía (usualmente energía eléctrica) y la convierte en algún tipo de movimiento (lineal, rotatorio u oscilatorio).

A continuación se citan algunos ejemplos de actuadores:

Módulo relé: su funcionamiento es el mismo que el de un interruptor, en este caso controlado por un circuito eléctrico compuesto por una bobina y un electroimán, que permiten el paso de electricidad a través de unos u otros contactos y de esta forma conseguir abrir o cerrar otros circuitos independientes conectados. Por lo tanto éste módulo permitirá activar otros actuadores independientes.

Figura 2.8. Modulo relé

Módulo LCD: aunque no se trate de un dispositivo mecánico que transforme la energía eléctrica en movimiento, éste módulo produce un cambio en el contexto de forma visual, por lo que puede resultar útil para visualizar información del objeto al que se encuentre conectado.

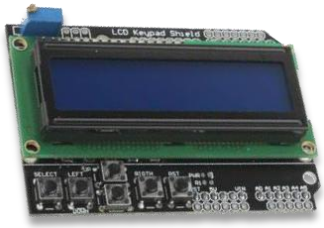

Figura 2.9. Módulo LCD

2.2.3. Conectividad: módulo de red ENC28J60

Características generales

Para que los objetos puedan comunicarse entre sí es imprescindible que dispongan de algún elemento de comunicación y así poder construir una red de objetos inteligentes

capaces de transferir información de su estado a otras entidades, ya sean otros objetos o servidores.

El presente proyecto está basado en la interconexión de una placa Arduino y un módulo de conectividad Ethernet ENC28J60, suponiendo una solución más económica frente al módulo de red original para Arduino Ethernet Shield W5100.

El ENC28J60 es un módulo de bajo coste debido principalmente a los componentes utilizados para su fabricación.

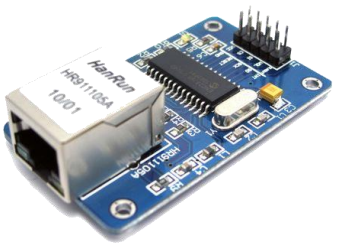

Figura 2.10. Módulo de red Ethernet ENC28J60

Éste módulo está basado en el chip ENC28J60 del fabricante Microchip y cumple con las especificaciones IEEE 802.3. Opera con una frecuencia de reloj de hasta 20 Mhz a través de la interfaz SPI, integra un controlador MAC y cuenta con un buffer gestionado a nivel de hardware de tamaño de 8 KB que se encuentra dividido en espacios de memoria separados para la recepción y la transmisión.

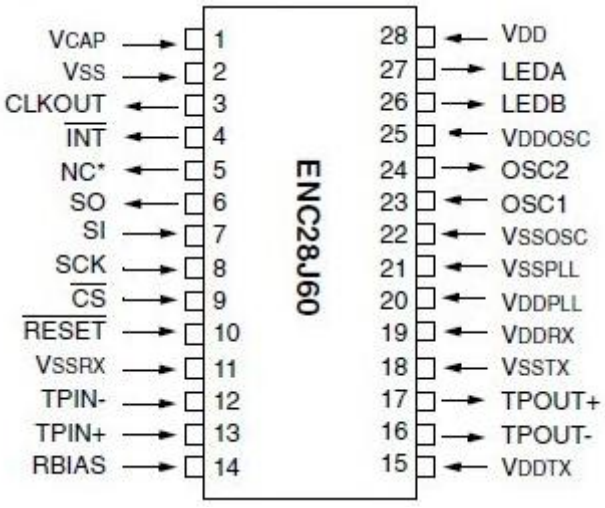

Figura 2.11. Pinout del chip ENC28J60

El tamaño y localización de dichos espacios de memoria, al igual que la retransmisión de datos en caso de colisión, son parámetros totalmente programables a través del microcontrolador utilizando comandos de la interfaz SPI.

El buffer de recepción consiste en un buffer circular FIFO (First In, First Out), y cualquier espacio dentro de los 8 KB de memoria que no está programado como parte del buffer de recepción, estará considerado como buffer de transmisión.

El controlador MAC soporta envíos de paquetes en modo Unicast, Multicast y Broadcast. Dispone además de un mecanismo WOL (Wake on Lan) para despertar de un estado *standby* tras la recepción de diversos formatos de paquetes (Magic Packet, Unicast, Multicast, Broadcast o por coincidencia específica de un tipo de paquete).

A pesar de que el ENC28J60 soporta transferencias Half-Duplex y Full-Duplex, la negociación automática de transmisión dúplex no es posible. Si el módulo se encuentra conectado a algún dispositivo habilitado para la detección y configuración automática de dúplex, se detectará como un dispositivo Half-Duplex por defecto.

Para establecer el modo de transmisión a Full-Duplex, ambos extremos deberán ser configurados manualmente. El control de flujo implementado por hardware está disponible para ambos modos de transmisión Half-Duplex y Full-Duplex.

Interfaz de comunicación con microcontrolador

La interfaz utilizada para conectar el módulo ENC28J60 a la placa Arduino es la SPI (Serial Peripheral Interface). El protocolo SPI permite la transmisión de datos en serie de forma síncrona, comúnmente utilizado para la comunicación entre microcontroladores y otros dispositivos periféricos en corta distancia. Puede ser utilizado también para comunicar dos microcontroladores.

La velocidad máxima de transferencia que se puede alcanzar mediante el bus SPI con el módulo ENC28J60 es de 10 Mbps.

En una conexión SPI, existe siempre un dispositivo maestro (usualmente un microcontrolador) el cual controla los dispositivos periféricos.

Alimentación

La alimentación que puede recibir el módulo para su funcionamiento puede ser de 3.3V o 5V.

Interrupciones

Cabe destacar que el ENC28J60 dispone de múltiples fuentes de interrupciones y un pin de salida para enviar una señal al microcontrolador ante cualquier ocurrencia de evento.

Este pin de interrupción está designado para ser utilizado por un microcontrolador que sea capaz de detectar flancos de bajada y deberá conectarse y configurarse adecuadamente en el microcontrolador Arduino Mega 2560 en cualquiera de los pines de interrupción (2, 3, 18, 19, 20 o 21).

3. Descripción de componentes Software

3.1. Programación sobre plataforma Arduino

La plataforma Arduino está basada en el lenguaje C y soporta todas las funciones del estándar C y C++. Dispone de una amplia diversidad de lenguajes de programación de alto nivel, además de los mencionados como C#, Java, Python, Perl, PHP, Visual Basic, Matlab, entre otros.

Los programas compilados con Arduino se enlazan contra la librería AVR Libc.

AVR Libc es un proyecto de software libre que tiene como objetivo el proporcionar una biblioteca C para ser utilizada con el compilador GCC sobre microcontroladores Atmel AVR. Consta de 3 partes fundamentales:

- avr-binutils: Es una colección de programas utilizados para manipular archivos objeto y binarios creados para la arquitectura Atmel AVR.
- avr-gcc: Compilador GNU C para arquitectura Atmel AVR.
- avr-libc: Contiene librerías estáticas y los archivos de cabecera requeridos en el proceso de compilación.

Algunas particularidades de la API de Libc AVR

La API de la librería AVR Libc dispone de algunos elementos de programación solamente disponibles en microcontroladores. A continuación se exponen algunos ejemplos:

Interrupciones: las señales de interrupción son las siguientes:

cli(): Desactiva las interrupciones globales (Clear Interrupts).

sei(): Activa las interrupciones globales (Set Interrupts).

Éstas instrucciones afectan al temporizador y a la comunicación en serie. La llamada a la función *delayMicroseconds()* desactivaría las interrupciones.

Temporizadores: la función *delayMicroseconds()* genera el menor retardo posible en Arduino y ronda los 2 microsegundos. Se pueden obtener retardos aún más pequeños con la instrucción en ensamblador *nop* (No Operation), cuya función es la de ejecutar un único ciclo de reloj. Teniendo en cuenta que el reloj del Arduino opera a 16 Mhz, la duración (período) de un ciclo de reloj es de aproximadamente 62.5 nanosegundos.

Manipulación de bits en variables: existen definiciones como *cbi* y *sbi* que son mecanismos estándar de AVR cuya utilidad es la de facilitar la manipulación de bits en variables, principalmente de tipo PORT.

Librerías

Como en todo lenguaje de programación, en Arduino también es posible la inclusión de librerías propias que proporcionen una funcionalidad extra y así favorecer el desarrollo modular de aplicaciones. Existen una serie de librerías estándar ofrecidas por la plataforma software Arduino.

En el caso de necesitar soporte para hardware adicional, hará falta incluir en el código la librería que dé la funcionalidad en concreto para dicho hardware.

Estructura de un programa en Arduino

La estructura básica de un programa en Arduino es simple y se compone de dos funciones:

```
void setup()  {
        < instrucciones >
}

void loop() {
        < instrucciones >
}
```

La función *setup()* es la encargada principalmente de la inicialización de variables para la configuración de dispositivos periféricos, puerto serie, modo de trabajo de los pines o cualquier otra variable, previamente a la ejecución de la función *loop()*.

Una vez ejecutada ésta función (sólo una vez), se pasará a la ejecución de la función *loop()*.

La función *loop()* es la función que contendrá el código del programa principal y se ejecutará cíclicamente como su nombre sugiere, de igual forma que lo hace un bucle infinito, respondiendo continuamente a los eventos que se produzcan en la placa.

Por otra parte, como se verá más adelante, el uso de un sistema operativo exige un leve cambio en la estructura básica de un programa en Arduino.

En el caso del uso de un sistema operativo como ChibiOS/RT, la estructura básica quedaría de la siguiente forma:

```
void setup() {
        < instrucciones >
        chBegin( mainThread );
}

void mainThread() {
        < instrucciones >
}

void loop() {
        // No se utiliza, pero debe declararse de todas formas.
}
```

Como se puede apreciar en el ejemplo anterior, el flujo de ejecución empieza desde la función *setup()* que a su vez llama a la función *chBegin(mainThread)*, que finalmente llama a la función *mainThread()*, donde reside el código del programa principal.

A pesar de este pequeño cambio en la estructura del código respecto al original, es necesario declarar la función *loop()* en cualquier caso. Cualquier instrucción que resida en el cuerpo del código de la función *loop()* será ignorada.

3.2. Entorno de desarrollo predeterminado

El entorno de desarrollo original de Arduino es una herramienta de código abierto, que permite escribir el código de programas conocidos como *sketch*, cargarlos en la placa Arduino y comunicarse con ellos.

Este entorno está compuesto por un editor de texto para escribir el código, dispone de una consola que proporciona texto de salida, mensajes de error e información adicional, una barra de herramientas con botones para las funciones comunes, y una barra de menú superior del cual cabe destacar las siguientes opciones:

- Fácil inclusión en el sketch de librerías: Sketch → Importar Librería[1]
- Fácil acceso a código de librerías y proyectos: Archivo → Sketchbook
- Fácil apertura de código que diversas librerías aportan a modo de ejemplo de uso: Archivo → Ejemplos

[1] Para que el entorno de desarrollo pueda encontrar las librerías, éstas deben residir en la carpeta de documentos de usuario "My Documents\Arduino\Libraries", donde cada subdirectorio representa una librería diferente y dentro del cual debe existir al menos un archivo de cabecera .h y otro de implementación .cpp. Tanto el directorio como los archivos .h y .cpp principales de la librería deben tener el mismo nombre. Es probable que haya que reiniciar el entorno de desarrollo para que sean accesibles las nuevas librerías instaladas.

- Comunicación con el programa previamente cargado en la placa Arduino: Herramientas → Puerto Serial

La principal finalidad de este entorno es la de facilitar la tarea al programador para desarrollar aplicaciones, ya que es intuitivo y proporciona lo necesario para poner en funcionamiento cualquier código en muy poco tiempo.

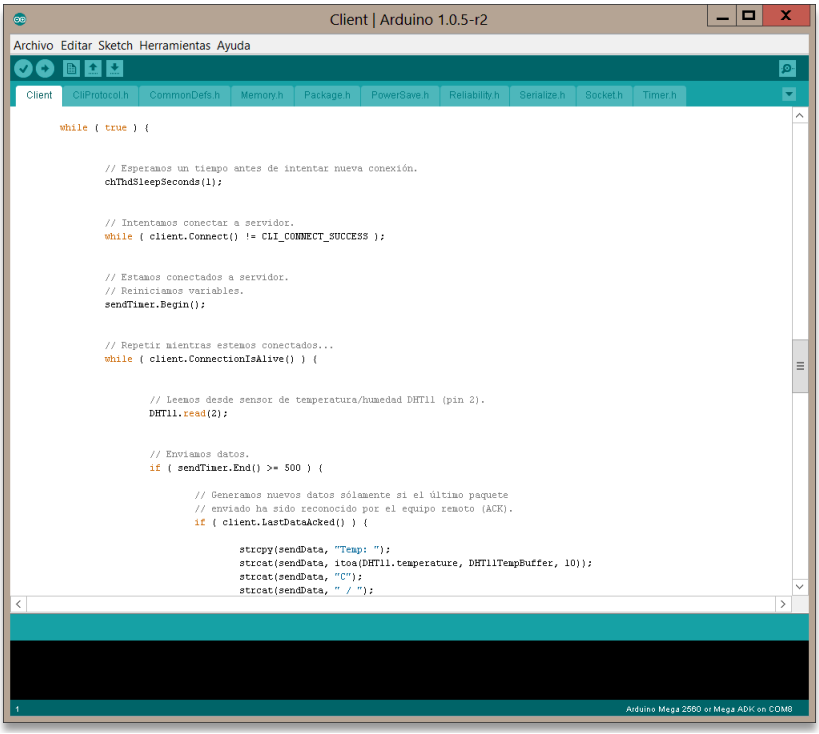

Figura 3.1. Entorno de desarrollo de Arduino

3.3. Visual Studio como entorno de desarrollo

La sencillez y facilidad de uso que aporta el entorno de desarrollo original para Arduino se traduce en cierta limitación en lo que respecta a la organización del sketch o código y la inclusión de librerías, como así también la depuración de código cuando éste adquiere una mayor complejidad.

En el caso de sistemas Windows, existe la posibilidad de utilizar Visual Studio como entorno de desarrollo. Para ello se requiere la instalación de un plugin conocido como Visual Micro.

A continuación se citan algunas características importantes a destacar de Visual Micro:

- La instalación del plugin configura Visual Studio automáticamente.

- Soporte para todas las versiones de Arduino como también otras placas compatibles, como Intel Galileo, Teensy, Attiny, avrIO, LaunchPad, ChipKIT/Pic32, STM32 y Texas Instruments StellarPad.

- Fácil inclusión de librerías.

- Rápida compilación de código.

- Eficiente monitor de puerto serie.

- Sistema de autocompletado *IntelliSense*, que supone una de las principales ventajas comparado al entorno de desarrollo original de Arduino. Este sistema permite la documentación y desambiguación de nombres de variable, funciones y métodos, lo que se traduce en una considerable ayuda para agilizar el desarrollo de código.

- Múltiples proyectos sketch en una única solución de Visual Studio.

- Total compatibilidad entre código del entorno original de desarrollo de Arduino y Visual Studio.

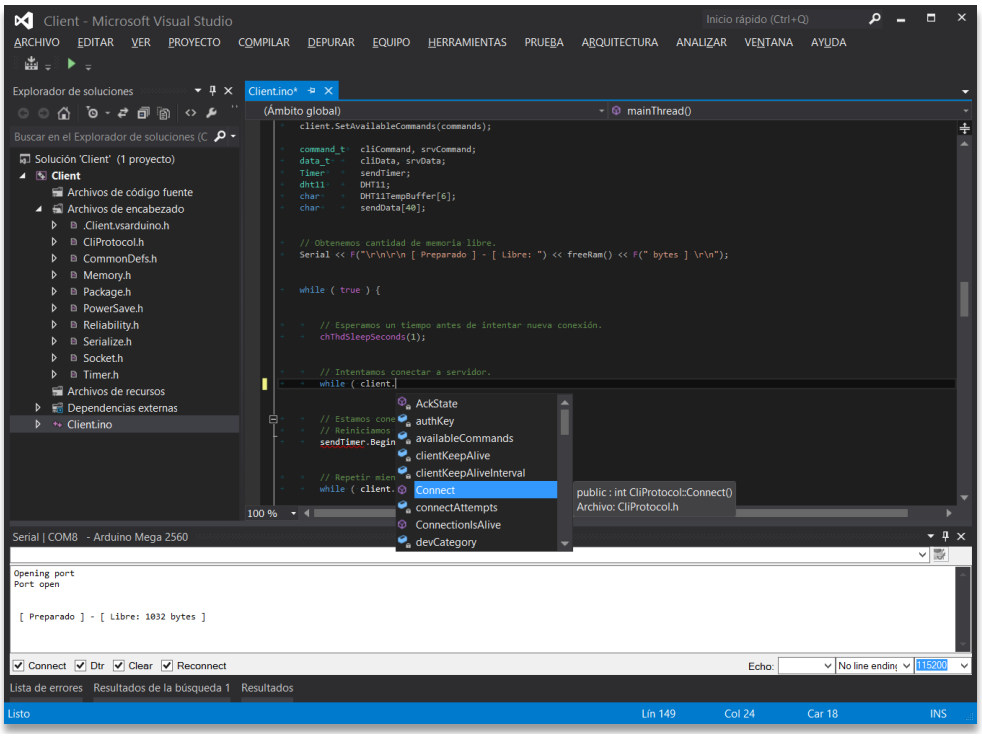

Figura 3.2. Entorno de desarrollo Visual Studio 2012

4. Tecnologías Empleadas

4.1 Sistema operativo de tiempo real ChibiOS/RT

El Arduino Mega 2560, al igual que el resto de microcontroladores Arduino, no cuenta con un sistema operativo que gestione los recursos hardware propios del microcontrolador, por lo tanto el código será ejecutado directamente sobre el microprocesador y no habrá límites de acceso al hardware más que el impuesto por el compilador y la funcionalidad que proporcionen las librerías de los dispositivos periféricos conectados a la placa.

El tipo de aplicación que se pretende diseñar para Internet de las Cosas es aquella que permita a un objeto obtener datos del contexto desde sensores para que posteriormente, dichos datos puedan ser procesados e interpretados por otro objeto.

La naturaleza de este tipo de aplicaciones exige la ejecución de tareas dentro de un intervalo de tiempo limitado, lo más preciso posible.

Para llevar a cabo tareas que puedan ser ejecutadas con fiabilidad y en un lapso de tiempo limitado, se utilizan los sistemas operativos de tiempo real o RTOS (Real Time Operating System).

En la actualidad los sistemas operativos de tiempo real están presentes en todo tipo de dispositivo empotrado, generalmente hardware específico donde exista un alto nivel de interacción con el entorno: en fábricas, para controlar un proceso de montaje o producción, decodificadores de televisión, sistemas de telecomunicaciones, radares, aviones, trenes, vehículos, aparataje sanitario, etc.

El Arduino Mega 2560 dispone de un único microprocesador mono núcleo, por lo tanto el uso de un RTOS no permitirá ejecutar tareas en paralelo, sino que las diversas tareas que se encuentren en ejecución serán alternadas según una pauta marcada por el planificador de tareas del RTOS.

La rápida alternancia de tareas (cambio de contexto) por parte del planificador dará la sensación de ejecución simultánea de todas las tareas.

A pesar de no existir una verdadera ejecución en paralelo, lo que sí aporta el uso de un RTOS son diversos mecanismos de sincronización, además de la posibilidad de establecer un orden preciso de ejecución y la asignación de diferentes prioridades entre tareas (una tarea que debe ser ejecutada en un tiempo crítico tendrá una mayor prioridad que el resto, con las que compite por la CPU).

La utilización de un sistema operativo de tiempo real como parte del desarrollo de aplicaciones para microcontroladores y en este caso, para Internet de las Cosas, supone un valor añadido dado que permite enriquecerlas con diversas metodologías basadas en técnicas de tiempo real, no disponibles por defecto con los elementos de programación y hardware básicos en la plataforma Arduino.

Características generales

ChibiOS/RT es un sistema operativo de tiempo real, de código libre, diseñado para ser utilizado en aplicaciones embebidas donde la eficiencia de ejecución y código compacto son requerimientos importantes.

Se caracteriza principalmente por un cambio de contexto muy eficiente, una alta capacidad de portabilidad, tamaño compacto y un completo conjunto de primitivas:

- Hilos
- Reserva y liberación de memoria segura (Thread-safe Memory Allocators)
- Temporizadores virtuales
- Mecanismos de sincronización de hilos:
  - Semáforos
  - Mutexes
  - Variables condición
- Otros mecanismos de sincronización de hilos:
  - Mensajes síncronos y asíncronos
  - Mailboxes
  - Flags de eventos
  - Colas de E/S

Planificador

ChibiOS/RT utiliza un *scheduler* o planificador de tareas preemptivo, es decir, hace las veces de árbitro y cede la CPU a la tarea de mayor prioridad, postergando la entrada en la CPU del resto de tareas hasta que la de mayor prioridad haya acabado su ejecución.

Si varias tareas tienen la misma prioridad, entonces el planificador cede la CPU a una de ellas según un algoritmo con estrategia de rotación *Round-Robin*.

Por otro lado también es posible configurar ChibiOS/RT en tiempo de compilación para que utilice una política de planificación cooperativa, es decir, cada tarea es responsable de ceder la CPU.

Quedaría por tanto en manos del programador el llamar explícitamente a la función necesaria para que cada tarea ceda la CPU, o de lo contrario el sistema se bloquearía.

Memoria

El kernel tiene un diseño estático, es decir, no se reserva ni se libera memoria en él. Esto le confiere más seguridad y robustez.

Habrá un sólo proceso en ejecución dividido en varias tareas, lo que significa que éste proceso compartirá el mismo espacio de memoria que el kernel y no existirá ninguna protección de la zona de memoria dedicada al mismo como sucede en otros sistemas operativos de propósito general.

Cabe remarcar la importancia de una adecuada gestión de la memoria en sistemas operativos de tiempo real, ya que generalmente el dispositivo deberá funcionar indefinidamente, sin necesidad de necesitar ningún reinicio.

No debe existir memoria reservada y que nunca fuera liberada y además la velocidad de reserva de memoria debería ser lo suficientemente alta como para cumplir con los requerimientos de tiempo de un sistema operativo de tiempo real.

Por estas razones, es preferible que toda la memoria requerida sea reservada en tiempo de compilación y no dinámicamente en tiempo de ejecución.

Otras características

Otros detalles no menos importantes y que actualmente no están implementados en su totalidad para la plataforma Arduino son:

- HAL (Hardware Abstraction Layer): Capa de abstracción de hardware, lo que proporciona portabilidad a diferentes plataformas.

- Soporte para componentes externos y otros subsistemas:
  - Pilas TCP/IP: uIP, lwIP
  - Sistemas de ficheros: FatFS

4.1.1. Métodos de E/S: Interrupciones y Polling

De forma predeterminada, para la recepción de paquetes se utiliza el mecanismo conocido como *polling*. Este método se basa en verificar continuamente una variable o *flag* que indica el estado de un evento (p. ej. de un periférico).

A continuación se presentará una comparación entre polling e interrupciones:

Polling:

- Se debe revisar continuamente si el periférico está preparado o no, lo que conduce a un uso intensivo de la CPU.
- El código es generalmente menos estructurado.
- El código tiende a inducir latencias variables en el servicio de periféricos.

Interrupciones:

- Utilizan la CPU solamente cuando hay trabajo que realizar.
- El código para un periférico en concreto suele concentrarse en un sólo lugar (ISR).
- Las latencias de servicio de periféricos son altamente predecibles.

Relación entre interrupciones y consumo energético

Al utilizar polling se genera una respuesta más rápida que las interrupciones, debido principalmente a que la atención de interrupciones es computacionalmente costosa ya que se requiere invertir tiempo en guardar el estado actual de ejecución para reanudarlo posteriormente.

A pesar de ello, el uso de interrupciones frente a polling tiene una notable ventaja y es la utilización de la CPU solamente cuando es necesario, permitiendo al dispositivo entrar en diversos modos de ahorro de energía y traduciéndose consecuentemente en un mayor tiempo de vida útil, así como en un mayor tiempo de funcionamiento de microcontroladores alimentados por baterías.

Los microcontroladores AVR pueden establecerse en diferentes estados de ahorro de energía y una vez el dispositivo ha entrado en uno de estos estados, la reactivación se produce mediante la llegada de una interrupción externa, por lo que deberá añadirse la capacidad de reactivar el dispositivo a partir de la recepción de paquetes de datos por parte el módulo de red.

En el presente proyecto se añadirá la opción de recepción de paquetes mediante el uso de interrupciones, aunque se podrá utilizar cualquiera de las dos técnicas descritas anteriormente de manera indiferente.

A continuación se explicará la relación existente entre las interrupciones y el uso de tareas de ChibiOS/RT y sus elementos de sincronización.

4.1.2. Sincronización entre interrupciones y tareas

Interrupciones: conceptos generales

Unos de los papeles más importantes que tiene un sistema operativo de tiempo real en un microcontrolador es el de la gestión de interrupciones, ya que son una importante fuente de eventos ante los cuales el sistema debe reaccionar, pudiendo proceder éstas desde una fuente interna (software) o externa (hardware).

En ambos casos, la interrupción alerta al procesador de una condición de alta prioridad: el procesador interrumpe la ejecución del código actual, guarda el último estado de ejecución, deshabilita las interrupciones y procede a la ejecución de un código conocido como manejador de interrupción o ISR (Interrupt Service Routine) para tratar el evento que ha dado lugar a la interrupción.

Una vez terminada la ejecución del ISR, el procesador restaura el estado que había guardado previamente, vuelve a habilitar las interrupciones y reanuda la ejecución del código suspendido.

En ocasiones es necesario compartir alguna variable entre el código del ISR y el código principal u otro código, con lo cual el acceso simultáneo a dicha variable para lectura y escritura puede ser inestable (ocurriría una condición de carrera).

Una solución a esto es la declaración de la variable compartida como *volatile*, con lo cual se indica al compilador que cuando cargue la variable desde memoria, la almacene nuevamente y de inmediato en la memoria después de realizar cualquier operación sobre ella.

Por otra parte, es muy importante que el código del ISR sea lo más corto posible (debe durar poco tiempo de ejecución) para poder volver a activar las interrupciones al salir del mismo y así poder atender otras nuevas interrupciones inmediatamente. Este es un aspecto a tener en cuenta, dado que en el procesador AVR y por tanto en Arduino, no es necesaria la inhibición y posterior habilitación de interrupciones explícitamente (llamando a funciones *cli()* y *sei()* respectivamente).

Esto significa que por defecto no está permitida la anidación de interrupciones, es decir, si suceden múltiples interrupciones al mismo tiempo éstas serán atendidas siguiendo un orden FIFO.

Consecuentemente, si dos o más interrupciones del mismo tipo suceden mientras se está ejecutando el código del ISR, se recibirá solamente una llamada más al ISR para esa interrupción una vez finalice su código actualmente en ejecución y el resto de interrupciones que se hayan podido ir acumulando se descartarán (se pierde la interrupción).

Sincronización entre interrupciones y tareas

Debido a la necesidad de ejecutar el ISR en el menor tiempo posible, se requiere de algún mecanismo que solamente invierta tiempo en enviar una señal para activar otro hilo o tarea responsable de gestionar el tratamiento de la interrupción.

Para conseguir dicho cometido, bastará con utilizar algún mecanismo de sincronización entre tareas, por ejemplo un semáforo, llamando a la función *chBSemSignalI()* desde el ISR, lo que enviará una señal al semáforo que se encuentra a la espera con la función *chBSemWait()* presente en otro hilo. Éste a su vez será el encargado de ejecutar el código de gestión de la interrupción.

Las secciones críticas en este hilo deberán estar protegidas del acceso concurrente por parte de otros hilos mediante mutexes, con las correspondientes llamadas a las funciones *chMtxLock()* y *chMtxUnlock()*.

Es importante remarcar que en un sistema en el que coexistan el uso de interrupciones y tareas o hilos, el procesador considerará más prioritaria a la interrupción de menor prioridad que a cualquier otra tarea o hilo de mayor prioridad.

4.2. El protocolo UDP

UDP es un protocolo de comunicaciones de la capa de transporte que a su vez utiliza el protocolo IP para la capa de red.

Se define en el RFC 768 y utiliza el mecanismo conocido como "entrega de mejor esfuerzo", es decir, la entrega de datos ocurre de la forma más rápida posible en detrimento de una garantía de entrega en destino, aunque sí implementa un control de errores de tramas que pudieran estar corruptas (*checksum*).

Comparado a otros protocolos como TCP, con UDP no se establecen conexiones de punto a punto entre los sistemas que se comunican.

El protocolo UDP no ofrece fiabilidad o garantía de entrega de mensajes (*reliability*) por lo que estos pueden llegar fuera de orden, repetidos o perderse.

Además, no dispone de control de flujo y permite el envío de datagramas con una velocidad máxima impuesta por la interfaz de red, en muchas ocasiones superior a lo soportado por los dispositivos de encaminamiento entre el origen y el destino.

A pesar de todos los factores mencionados anteriormente, UDP ofrece comunicaciones muy eficientes para determinadas aplicaciones, los cuales habrá que tener en cuenta diseñando la aplicación responsablemente.

4.2.1. Utilización de UDP en Internet de las Cosas

El protocolo UDP dispone de características que lo hacen más apropiado para ser utilizado en aplicaciones que no requieran un uso excesivo de memoria, por ejemplo aquellas presentes en microcontroladores, como también en el caso de requerir el envío de información y no represente una penalización el hecho de perder parte de la misma.

El protocolo UDP permite el envío de paquetes de la forma más rápida posible. Se trata de un protocolo sin conexión, es decir, a diferencia de TCP que mantiene las conexiones activas por dirección IP y puerto para así poder transferir un flujo de paquetes, UDP permite el envío de mensajes a una dirección IP y puerto de destino sin mantener una conexión persistente.

Este hecho supone una ventaja de UDP frente a TCP en microcontroladores, ya que TCP requiere mantener abierta la conexión con el equipo remoto y almacenar en memoria información referente a la fiabilidad del protocolo (retransmisión de paquetes en caso de pérdida o reordenación de los mismos en caso de recepción de mensajes fuera de orden) y control de flujo y de congestión.

Este hecho supone un mayor consumo de memoria como también un mayor tiempo de cómputo invertido por algoritmos que no son necesarios en UDP.

Otra de las ventajas más importantes del uso de UDP frente a TCP es un menor tamaño de cabecera por paquete, siendo para UDP unos 8 bytes y para TCP unos 20 bytes.

Este es un factor que toma importancia cuando se requiere el envío constante de información, a una alta frecuencia de envío de paquetes, lo que evidentemente supondría una mayor sobrecarga y uso de ancho de banda por parte de TCP. Por ejemplo, si se envían 70 bytes utilizando UDP, el tamaño total del paquete son 98 bytes (ya que se suman 20 bytes por la cabecera IPv4 y 8 bytes por la cabecera UDP), lo que supone una sobrecarga de 30%; si se envían 150 bytes con UDP, el tamaño total del paquete serían 178 bytes, lo que se traduce en una sobrecarga de 15%.

En cambio, si se utilizara TCP para enviar las anteriores cantidades de bytes, la sobrecarga representaría un 40% y un 24% respectivamente.

Sin embargo, UDP tiene ciertas desventajas, principalmente en lo que respecta a la fiabilidad en el envío de paquetes.

Con el protocolo UDP, la recepción de mensajes por parte del equipo remoto no se controla con ningún tipo de acuse de recibo, ni tampoco se mantiene un control en el orden de recepción de los mensajes. Por lo tanto, cualquier pérdida de mensajes o recepción fuera de orden no será verificado y no se realizará ningún control para su retransmisión.

A falta de un sistema de retransmisión de mensajes, habrá que tener presente otro factor importante que es la fragmentación de paquetes. Por este motivo, es imprescindible que el envío de información se realice en paquetes con un tamaño nunca superior a la unidad máxima de transferencia (Maximum Transfer Unit o MTU), parámetro establecido por los dispositivos de red que intervienen en el encaminamiento de paquetes.

En caso de ser necesario agregar algún tipo de control de fiabilidad parcial y así poder aprovechar las ventajas del protocolo UDP, es muy conveniente hacerlo en la capa de aplicación, como se verá más adelante.

4.2.2. Otros protocolos basados en UDP para Internet de las Cosas

Existen en la actualidad diversas propuestas para el desarrollo de un protocolo de alto nivel que llegue a ser estandarizado y utilizado por dispositivos que se conecten a Internet de las Cosas y que además utilicen el protocolo UDP como capa de transporte.

Uno de los más importantes es el CoAP (Constrained Application Protocol).

Actualmente existe un borrador de la propuesta de estandarización de CoAP por parte de la Organización de Estándares de Protocolos de Internet (Internet Engineering Task Force o IETF), que se puede consultar en https://datatracker.ietf.org/doc/draft-ietf-core-coap.

En este borrador se sugiere la utilización de CoAP en dispositivos electrónicos con recursos hardware muy limitados como son los microcontroladores, con una cantidad de memoria escasa y bajo consumo energético.

Este protocolo está basado en la arquitectura REST (Representational State Transfer) que permite identificar dispositivos mediante lo que se conoce como URI (Universal Resource Identifier) y hace uso del protocolo HTTP para controlar abstracciones de recursos a través de un servidor, utilizando las conocidas directivas GET, POST, PUT, etc.

Aunque CoAP es un buen candidato a ser elegido como protocolo estándar para Internet de las Cosas, a día de hoy las implementaciones del mismo requieren más memoria y capacidad de procesamiento que la disponible en la mayoría microcontroladores de 8 y 16 bits.

4.3. El protocolo TFTP

TFTP o Trivial File Transfer Protocol es un protocolo de transferencia de datos que se caracteriza por su simplicidad y es semejante a una versión básica del conocido

protocolo de transferencia de ficheros FTP. Está definido por el RFC 1350 y utiliza el protocolo UDP como transporte.

Se suele implementar con una cantidad de memoria muy reducida, por lo que frecuentemente se utiliza para automatizar la transferencia de ficheros de configuración, como también el firmware en dispositivos como routers que no suelen requerir una unidad de almacenamiento.

Este protocolo sólo permite la lectura y escritura de ficheros desde o hacia el servidor remoto y no permite listar directorios ni requiere autentificación alguna. Además, se suele utilizar el puerto 69 a la escucha de peticiones de descarga de ficheros.

Cuando una petición es aceptada, el fichero solicitado se envía en bloques de 512 bytes (un sólo bloque por paquete), a excepción del último que puede ser menor de 512 bytes, indicando el fin del fichero (aunque si fuera de 512 bytes, se enviaría por último un paquete con 0 bytes de datos para indicar el fin del fichero).

Cada bloque enviado debe ser reconocido con un acuse de recibo (*acknowledgement* o ACK) por el equipo remoto antes de enviar el siguiente, por lo que el equipo que envía el fichero debe mantener una numeración de los bloques enviados. Cuando se envía un paquete se inicia un temporizador y transcurrido un determinado timeout, si no se ha recibido un ACK para dicho paquete, entonces se considera como perdido y por lo tanto se debe reenviar.

4.3.1. Utilización de TFTP para actualización remota de firmware

Independientemente de que los diferentes objetos se encuentren distribuidos en una red local o bien conectados directamente a Internet, es decir, con independencia del tamaño de la red donde se encuentren conectados, es importante dotar a los mismos de la capacidad de actualizarse sin requerir el desplazamiento por parte del personal encargado de la instalación y/o mantenimiento del firmware actual.

Añadir esta característica a los objetos conectados a la red no sólo representaría un ahorro en costes de desplazamiento, sino que supondría una ventaja en aquellos casos en los que los objetos a actualizar no fueran fácilmente accesibles de forma física. En el caso de que la actualización fuera requerida por una cantidad importante de objetos, la actualización remota tomaría una mayor relevancia.

Para que el microcontrolador que da funcionalidad al objeto en cuestión tenga la capacidad de actualizarse remotamente, es necesario la reprogramación del gestor de arranque, generalmente conocido como *bootloader*, cuya modificación es necesaria ya que lo que se desea es que al poner en marcha el dispositivo, éste pueda comparar y verificar la versión del firmware actualmente instalado contra otra versión disponible en algún servidor de firmwares.

La motivación en la elección del protocolo TFTP para la obtención de firmwares se debe principalmente a la simplicidad del mismo, ya que se trata de un protocolo que ofrece lo mínimo indispensable para descargar ficheros. Por lo tanto, el bootloader debe soportar UDP como protocolo de transporte y sobre éste, el protocolo TFTP.

4.3.2. Actualización del bootloader para soporte de protocolo TFTP

El gestor de arranque o bootloader es un programa básico que en el caso de microcontroladores AVR se ejecuta inmediatamente después de poner en funcionamiento o reiniciar el dispositivo, esto es, se trata del primer programa que se ejecuta. Tanto el bootloader como la aplicación principal residen en diferentes zonas predefinidas de la memoria flash del microcontrolador.

Su principal finalidad es la de inicializar registros y en el caso de Arduino, habilitar e inicializar el puerto USB entre otros (que posteriormente permitirá subir a la memoria flash la aplicación que se haya compilado).

Al finalizar este breve proceso, la ejecución saltará desde el bootloader hasta la aplicación principal que se encuentre en ese momento en la memoria flash.

Las placas microcontroladoras Arduino disponen de un bootloader instalado de fábrica, que permite subir fácilmente (grabar en la memoria flash) la aplicación compilada, a través del puerto serie o USB, desde el entorno de desarrollo.

Las diferentes versiones de bootloaders para los diversos tipos de placa Arduino se encuentran disponibles tanto en código fuente como en versión compilada en la distribución software de Arduino. El protocolo que incluye el bootloader y que permite grabar la aplicación compilada en la memoria flash, es el conocido como stk500 (stk500v2 para Arduino Mega 2560).

A pesar del soporte USB que ofrece el bootloader por defecto para programar la memoria flash, si se desea substituir el presente por otro más actual o que incluya otras características (como es el caso del que da soporte para la actualización del firmware desde un servidor TFTP), es necesario la utilización de un programador externo que utilice el conector ICSP (un ejemplo sería el programador USBAsp).

5. Diseño e Implementación

5.1. Diseño

En este capítulo se procederá a explicar los detalles de los principales componentes del proyecto, tanto hardware como software cuya programación se describirá más adelante con la implementación.

5.1.1. Diseño del hardware

En el diseño del sistema se han elegido como principales elementos hardware, una placa microcontroladora Arduino Mega 2560, un módulo de red ENC28J60 y otros elementos de fácil configuración, como un sensor de temperatura y humedad.

Adicionalmente, una pantalla LCD y un actuador relé, han servido de ayuda para el desarrollo del proyecto.

A continuación se muestra el esquema de cableado necesario para el montaje del sistema:

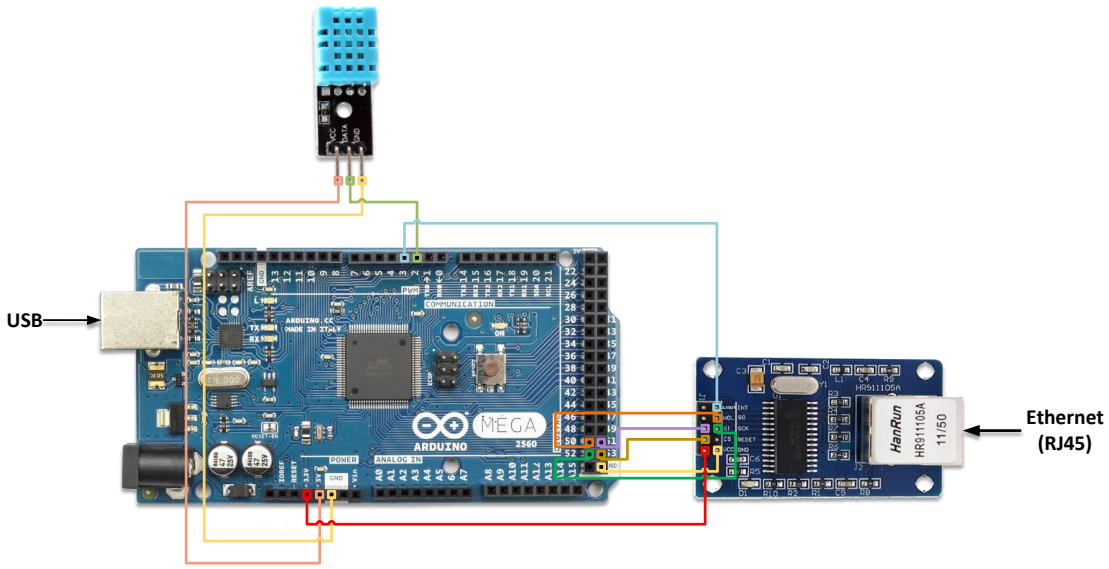

Figura 5.1. Esquema del montaje Arduino-ENC28J60

El detalle de la asignación de pines de todo el conjunto, se muestra a continuación en la siguiente tabla:

Arduino Mega 2560	Módulo ENC28J60	Módulo DHT11
50 (MISO)	SO	-
51 (MOSI)	SI	-
52 (SCK)	SCK	-
53 (SS)	CS	-
3 (INT 5)	INT	-
3V3 (3.3V)	VCC	-
GND	GND	GND
2 (INT 4)	-	DATA
5V	-	VCC

5.1.2. Diseño del software

En lo que respecta al software utilizado, se ha requerido una selección del mismo para los elementos acoplables y extensión de funcionalidades. Son los que se detallan a continuación:

- ChibiOS/RT: Sistema operativo de tiempo real para microcontroladores AVR.
  https://code.google.com/p/rtoslibs/downloads/list
  https://github.com/ChibiOS/ChibiOS-RT

- EtherCard: Librería para el módulo de red ENC28J60. Existen diversas implementaciones de la librería de libre distribución, habiéndose elegido una ya utilizada en múltiples proyectos.
  https://github.com/jcw/ethercard

- DHT11: Librería para el sensor de temperatura y humedad DHT11.
  http://playground.arduino.cc/main/DHT11Lib

- LiquidCrystal: Librería para el módulo LCD.
  http://arduino.cc/es/Tutorial/LiquidCrystal

- Streaming: Librería que proporciona una abstracción de funciones de salida por puerto serie y que permite utilizar el operador '<<' al estilo de *cout* para C++.
  http://arduiniana.org/libraries/streaming

- AVR-Etherboot: Bootloader para la familia de microcontroladores AVR. Utilizado como base para las ampliaciones realizadas sobre el mismo y dar soporte de control de versiones de firmware en red local o Internet.
  https://code.google.com/p/avr-etherboot

Adicionalmente, la contribución de código propio en el presente proyecto y que se respalda en las anteriores librerías, está compuesto de varios módulos en C++ y se estructura de la siguiente manera:

Cliente:

- CliProtocol.h: Protocolo de conexión del cliente.
- Client.ino: Programa principal de aplicación cliente.

Servidor:

- SrvProtocol.h: Protocolo de conexión y gestión de clientes (servidor).
- Server.ino: Programa principal de aplicación servidor.

Cliente / Servidor:

- CommonDefs.h: Parámetros del protocolo de conexión y tipos de mensajes (paquetes) de intercambio entre cliente y servidor, con sus respectivas operaciones de empaquetado y desempaquetado.
- Memory.h: Obtención de memoria disponible (memoria libre en el sistema).
- Package.h: Operaciones para facilitar el empaquetado y desempaquetado de datos a ser enviados y recibidos respectivamente.
- PowerSave.h: Activación de modo de ahorro de energía.
- Reliability.h: Sistema de fiabilidad para protocolo UDP.
- Serialize.h: Serialización utilizada en los procesos de empaquetado y desempaquetado de datos.
- Socket.h: Abstracción de operaciones de envío y recepción de paquetes. Gestión de paquetes recibidos a través de buffer circular.
- Timer.h: Definición de temporizador utilizado para calcular tiempo transcurrido entre diversas operaciones.

5.1.2.1. La clase Socket

La clase *Socket* es la clase fundamental utilizada tanto por el cliente como por el servidor para las operaciones de envío y recepción de paquetes UDP.

Los principales motivos para el diseño y desarrollo de ésta clase han sido:

- Presentar una alternativa al alto consumo de memoria por parte de la implementación existente para TCP.

- Superar el máximo número de conexiones permitido que ofrece la implementación existente para TCP.

- Añadir la posibilidad de sockets bloqueantes y no bloqueantes.

- Favorecer la recepción asíncrona de paquetes (dado que no siempre es posible leer el paquete en el preciso momento de la recepción), es decir, postergar la lectura del paquete recibido. Esto se puede conseguir con la utilización de buffers circulares.

En el diagrama que se presenta a continuación se detalla el diseño elegido para la implementación de la clase Socket basada en buffers circulares, tanto para el uso del método polling como mediante el uso de interrupciones:

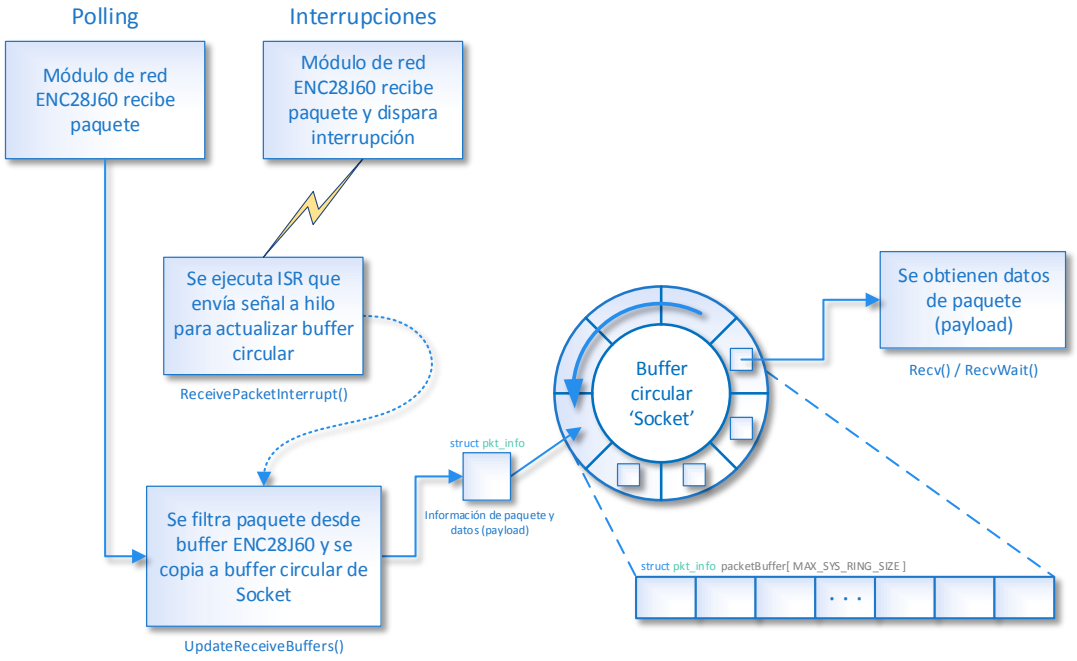

Figura 5.2. Diagrama de buffer circular

5.1.2.2. El cliente

El diseño de la aplicación cliente se apoya principalmente en la clase CliProtocol.

Cuando se desea desarrollar una aplicación que necesite establecer una conexión con otro cliente o servidor, se utilizará la clase desarrollada para facilitar todas las operaciones de comunicación necesarias.

Desde la puesta en marcha del dispositivo que ejecuta la aplicación cliente, se realizarán las siguientes operaciones:

- El hilo principal crea otro hilo independiente encargándose constantemente de la atención de paquetes recibidos y la gestión de los buffers circulares.
- Se valida el nombre de host o IP de destino.
- Se inicializan parámetros de cliente, tanto de conexión como información adicional que se transferirá al servidor.
- Se intenta establecer una conexión con el servidor.
- Se generan y formatean datos obtenidos desde sensores y cualquier otro dato adicional y se envían al servidor.
- Se verifica la disponibilidad de algún paquete recibido en el buffer circular, se procesa y ejecuta la operación correspondiente (por ejemplo, sobre un actuador si se trata de un paquete de comando).

A continuación se expone el diagrama de flujo completo correspondiente a la aplicación cliente:

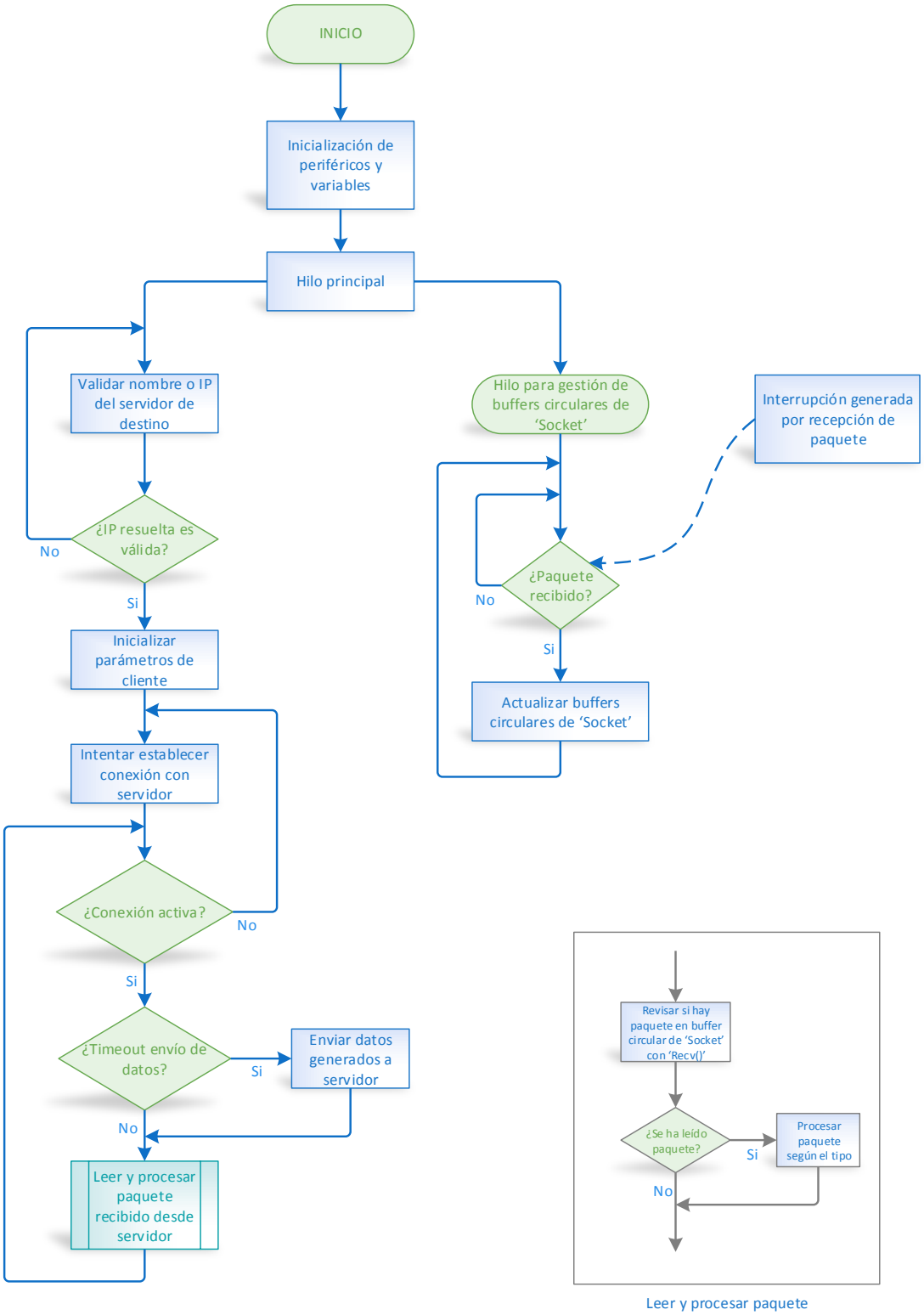

Figura 5.3. Diagrama de flujo de cliente

5.1.2.3. El servidor

El diseño de la aplicación servidor se apoya principalmente en la clase SrvProtocol.

El principal motivo del servidor es la de ofrecer a los clientes un punto de encuentro entre sí. Esto supone un beneficio cuando lo que se desea es conectar varios dispositivos en una red y que todos ellos fueran alcanzables desde Internet, lo cual requeriría la redirección de puertos (NAT) en la puerta de enlace de la red interna hacia el exterior para cada uno de los dispositivos cliente.

En cambio, si varios dispositivos cliente se conectaran a un servidor en común (en la misma red), sólo haría falta la redirección del puerto a la escucha del servidor en la puerta de enlace de la red interna, permitiendo no sólo a los dispositivos comunicarse entre sí, sino que también otros podrían conectarse al servidor desde el exterior (Internet).

Por otra parte, la existencia de un dispositivo servidor proporcionaría un comportamiento autoritativo si fuera requerido. Esto quiere decir que el servidor podría arbitrar y decidir qué comandos intercambiar y entre qué clientes, en base a cualquier tipo de regla predefinida.

A cada cliente conectado se le asigna un ID único basado en su IP y puerto de origen, luego, si un cliente quisiera comunicarse con otro, lo haría a través del servidor especificando el ID del cliente de destino. Es decir, la forma de direccionamiento no se realizaría de la forma tradicional por IP y puerto de destino (ya que podrían ser desconocidos), sino que se realiza a través de IDs.

Las peticiones que un cliente puede solicitar al servidor son las siguientes:

- Obtención de lista de IDs de clientes conectados.
- Obtención de parámetros de información de cliente conectado.
- Obtención de lista de comandos disponibles para cliente conectado.
- Obtención de muestra de últimos datos enviado desde de un cliente al servidor.
- Envío de comandos a otro cliente conectado.

Desde la puesta en marcha del dispositivo que ejecuta la aplicación servidor, se realizarán las siguientes operaciones:

- El hilo principal crea otro hilo independiente encargándose constantemente de la atención de paquetes recibidos y la gestión de los buffers circulares.

- Se inicializan parámetros de servidor.
- Se verifica la disponibilidad de algún paquete recibido en el buffer circular, se procesa y ejecuta la operación correspondiente, que puede ser:
  - Petición y establecimiento de conexión desde cliente.
  - Respuesta al cliente con petición solicitada.
  - Actualización de última muestra de datos recibida desde cliente.
  - Envío de comando a otro cliente.
- Se ejecutan otras tareas[2] adicionales, en el siguiente orden:
  - Se verifica caducidad de temporizador para entrar en modo de ahorro de energía (si no se ha recibido ningún paquete desde ningún cliente transcurrido un tiempo concreto).
  - Se verifica caducidad de temporizador para envío de mensajes de actividad del servidor a todos los clientes conectados.
  - Se verifica caducidad de temporizador para rastrear y marcar clientes como desconectados y liberar recursos si hiciera falta (si no se ha recibido ningún paquete desde ningún cliente transcurrido un tiempo concreto).

A continuación se expone el diagrama de flujo completo correspondiente a la aplicación servidor:

[2] Cabe mencionar que la motivación sobre el diseño basado en tareas independientes es la posibilidad (en futuras ampliaciones) de paralelizar la ejecución de las mismas en diferentes hilos, además de facilitar el trabajo de depuración del código.

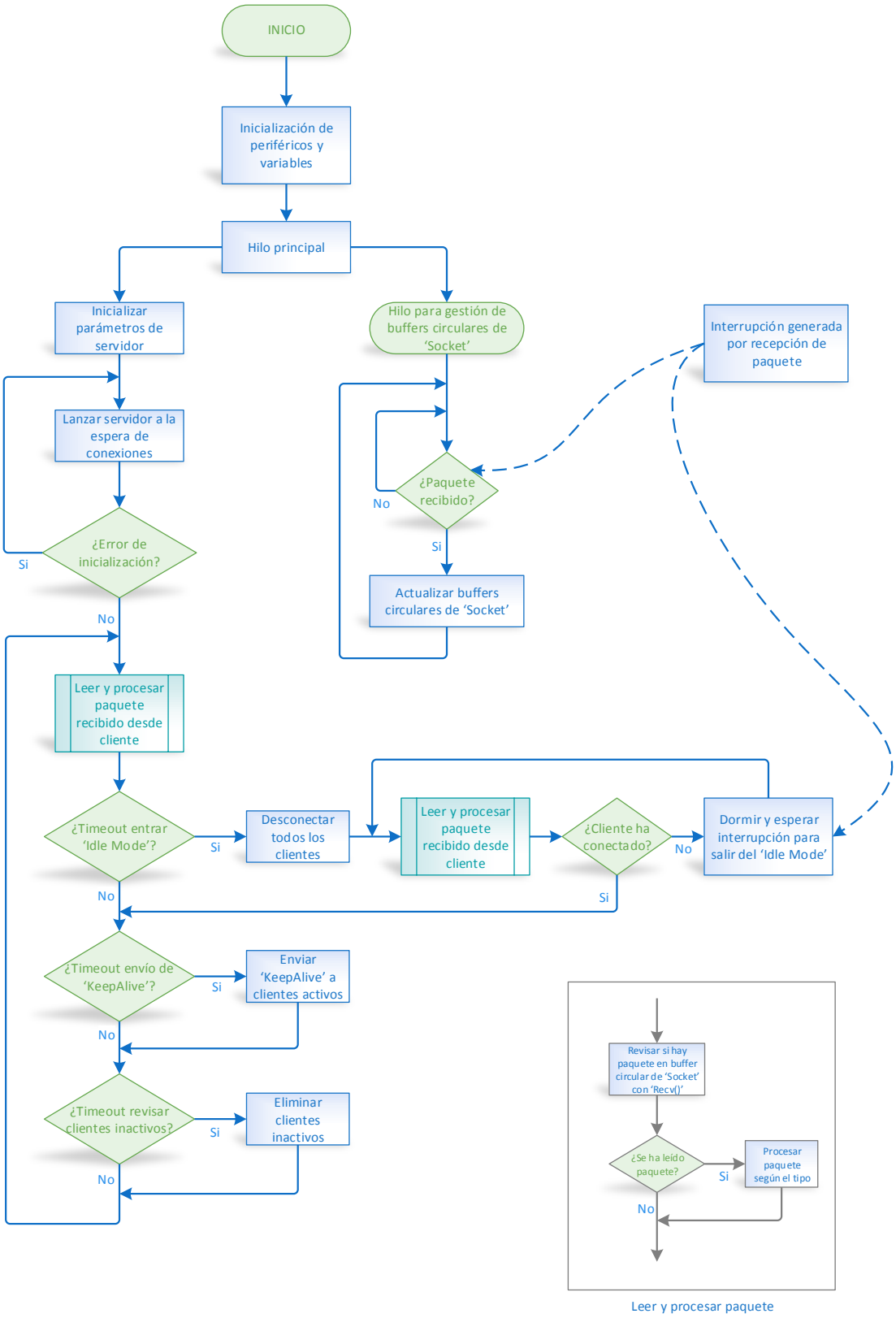

Figura 5.4. Diagrama de flujo de servidor

5.2. Implementación

Tanto la aplicación cliente como la aplicación servidor, comparten el mismo código en el proceso de puesta en marcha del dispositivo e inicialización de parámetros, con la misma estructura para ambos.

Como se ha mencionado anteriormente, el punto de partida de toda aplicación en Arduino es la función *setup()* seguida por la función *loop()*, la cual se ejecuta indefinidamente.

En el caso de trabajar con ChibiOS/RT para Arduino, se llamará en primer lugar a la función *setup()* en donde se inicializarán los periféricos, en este caso el puerto serie (velocidad) y la configuración de red (asignación de IP local y puerta de enlace si la configuración es estática o bien se procederá a la asignación automática de los parámetros desde un servidor DHCP disponible en la red).

El siguiente paso es la inicialización de ChibiOS/RT, que provocará el salto del flujo del programa desde *setup()* hacia *mainThread()* el cual se convertirá en el principal hilo de ejecución.

La función *mainThread()* creará un hilo de ejecución independiente mediante la llamada a la función *chThdCreateStatic()* e inmediatamente después llamará a la función *attachInterrupt()* para atender a las interrupciones generadas por recepción de paquetes por parte del módulo de red ENC28J60 (aunque existe la posibilidad de utilizar polling y no interrupciones, siempre y cuando no esté definida la variable 'USE_INTERRUPTS', lo cual es recomendable en caso de recibir una alta tasa de paquetes por segundo).

El nuevo hilo creado llamará a la función *ReceivePacketThread()* que se ejecutará indefinidamente, estableciendo un semáforo a la espera de una señal; cuando se desbloquee el semáforo, se llamará a la función *Socket::UpdateReceiveBuffers()*, cuya finalidad es la actualización de los buffers circulares de recepción.

A partir de este punto se detallarán las implementaciones por separado para el cliente como para el servidor, el sistema de fiabilidad en UDP utilizado para la comunicación de datos y finalmente, el sistema de control de versiones de firmware integrado en el bootloader.

5.2.1. El cliente

En primer lugar, el hilo principal o *mainThread()* entrará en un bucle infinito para intentar resolver el nombre de host al que se desea conectar.

Si se ha conseguido validar el nombre de host o bien se ha especificado una IP del servidor directamente, entonces se creará una instancia de cliente (clase CliProtocol) y se inicializarán los parámetros necesarios (IP y puerto de servidor).

En este punto también se inicializan otras variables (información del cliente que se envía al servidor en el momento del establecimiento de la conexión), llamando a los siguientes métodos de la instancia de cliente creada previamente:

Método (establece parámetro)	Tipo de parámetro
SetKeepAliveInterval(interval)	Frecuencia de envío de mensajes Keep-Alive
SetDeviceCategory(category)	Categoría del dispositivo
SetDeviceVersion(version)	Versión de firmware del dispositivo
SetAvailableCommands(commands)	Comandos disponibles de cliente

Posteriormente se ejecutará el bucle principal del cliente simulando el comportamiento de la función *loop()*.

En cada iteración se llamará al método *Connect()* que será el responsable de intentar establecer una conexión con el servidor. Por lo tanto, cada intento fallido de conexión irá seguido de infinitos intentos, hasta poder establecer una conexión exitosa.

Esto se verifica llamando al método *ConnectionIsAlive()* que es el encargado de comprobar constantemente si la conexión con el servidor sigue activa.

Dependiendo del comportamiento que se le quisiera dar al cliente, mientras la conexión permanezca activa, se podrá programar el envío de datos obtenidos a partir de sensores o enviar comandos y recibir datos o comandos para disparar en consecuencia algún evento que active algún actuador.

Si se envían paquetes de datos, queda en manos del programador verificar si el último paquete de datos ha sido reconocido por el servidor, llamando al método *LastDataAcked()*. Esto ofrece la flexibilidad para decidir si se desea enviar una nueva muestra de datos o volver a reenviar la anterior.

El pseudo-código quedaría de la siguiente forma:

```
if ( LastDataAcked() ) {

        nuevos_datos = < generar_nuevos_datos >

        SendData( nuevos_datos )

}

else {
        SendData( anteriores_datos )
}
```

A continuación se detallan los métodos accesibles (públicos) junto con sus parámetros de la clase CliProtocol:

Método	Descripción
CliProtocol(const char *host, uint16_t port)	Constructor
int Connect()	Intenta establecer una conexión con el servidor
bool ConnectionIsAlive()	Verifica si la conexión con el servidor sigue estando activa
void SetAttempts(uint16_t attempts)	Establece el número máximo de intentos de conexión con el servidor
void SetRetries(uint16_t retries)	Establece el número máximo de reintentos entre estados de *handshake* con el servidor
void SetRetryDelay(uint16_t timeout)	Establece el tiempo límite de espera de respuesta de conexión desde servidor
void SetKeepAliveInterval(uint32_t interval)	Establece la frecuencia de envío de mensajes Keep-Alive
void SetDeviceCategory(uint16_t category)	Establece la categoría del dispositivo
void SetDeviceVersion(uint32_t version)	Establece la versión de firmware del dispositivo
void SetAvailableCommands(command_list_t &commands)	Establece los comandos disponibles de cliente
void SendCommand(command_t &command)	Envía paquete de comando
int RecvCommand(command_t &command)	Verifica si se ha recibido algún paquete de comando
void SendData(data_t &data)	Envía paquete de datos
int RecvData(data_t &data)	Verifica si se ha recibido algún paquete de datos
void SendKeepAlive()	Envía mensaje Keep-Alive
int RecvKeepalive()	Verifica si se ha recibido algún mensaje de tipo Keep-Alive
bool LastDataAcked()	Verifica si el último paquete de datos ha sido reconocido por el servidor

5.2.2. El servidor

El hilo principal o *mainThread()* creará una instancia de servidor (clase SrvProtocol) e inicializará los parámetros necesarios (puerto a la escucha y máximo número de clientes permitidos) y ejecutará un bucle infinito, simulando el comportamiento de la función *loop()*.

Se llamará al método *Start()* de la instancia de servidor creada previamente y si no se produce ningún error de inicialización, se ejecutará el bucle principal del servidor para atender conexiones y peticiones de los clientes.

El bucle principal del servidor es el encargado de ejecutar periódicamente algunas tareas bien definidas y de forma secuencial.

En cada iteración del bucle, se llamará al método *ReceivePackets()* que como indica su nombre, es el encargado de revisar la disponibilidad de paquetes en el buffer circular del socket y procesarlos.

ReceivePackets() llamará a su vez a alguno de los siguientes métodos dependiendo del tipo de paquete recibido:

Método (procesa paquete)	Tipo de paquete
ProcessConnectRequestPacket()	Petición de conexión
ProcessConnectConfirmPacket()	Confirmación de conexión
ProcessDataPacket()	Datos
ProcessCommandPacket()	Comando
ProcessKeepAlivePacket()	Mensaje Keep-Alive

La ejecución del resto de tareas está gobernada por sus respectivos temporizadores, lo cual otorga la flexibilidad de modificar la frecuencia de ejecución de las mismas, con sus métodos correspondientes. Son los que se mencionan a continuación:

Método (establece parámetro)	Tipo de parámetro
SetSleepModeTimeout(timeout)	Frecuencia de llamada a tarea de gestión de ahorro de energía
SetKeepAliveInterval(interval)	Frecuencia de envío de mensajes Keep-Alive
SetInactivityCheckTimeout(timeout)	Frecuencia de llamada a tarea de gestión de clientes inactivos

La tarea de gestión de energía entra en ejecución cuando no se hayan recibido paquetes desde ningún cliente una vez ha transcurrido un cierto lapso de tiempo.

Al ejecutarse, en primer lugar, se reiniciará el contador de clientes activos (eliminándose de la tabla de clientes activos si es necesario) con el método *DisconnectClients()*.

Posteriormente se llamará al método *EnterLowPower()*. Éste método ejecuta un bucle, llamando en primer lugar a *SetLowPower()* y entrando en modo de ahorro energético.

Dado que una interrupción provocada por la recepción de un paquete de cualquier tipo provocaría la salida del modo de ahorro de energía, el bucle debe controlar el contador de clientes activos, lo cual se traduce en la salida del estado de ahorro energético solamente en el momento en que el primer cliente ha conseguido establecer una conexión, de lo contrario se volvería a llamar a *SetLowPower()*, poniendo nuevamente el dispositivo a dormir.

La tarea de envío de mensajes Keep-Alive entra en ejecución al llamar al método *SendKeepAlive()*, lo que provoca el envío de un mensaje de actividad del servidor a todos los clientes activos.

Por otra parte, la tarea de gestión de clientes inactivos se ejecuta al llamar al método *InactivityCheck()*, lo que provoca la verificación de clientes desde los cuales no se esté recibiendo información (paquetes de datos, comandos o mensajes Keep-Alive) una vez transcurrido un determinado período de tiempo, parámetro modificable mediante el método *SetMaxInactivityTime()*.

Seguidamente, se detallan los métodos accesibles (públicos) junto con sus parámetros de la clase SrvProtocol:

Método	Descripción
SrvProtocol(uint16_t port, uint16_t cli)	Constructor
int Start()	Iniciación de servidor
void SetMaxInactivityTime(uint32_t timeout)	Establece el tiempo límite para considerar a los clientes como inactivos
void SetKeepAliveInterval(uint32_t interval)	Establece el tiempo límite de envío de mensajes Keep-Alive a clientes
void SetInactivityCheckTimeout(uint32_t timeout)	Establece el tiempo límite de chequeo de clientes inactivos
void SetSleepModeTimeout(uint32_t timeout)	Establece el tiempo límite para entrar en *sleep mode*

5.2.3. Fiabilidad en UDP

La problemática a resolver se centra en tres situaciones que pueden darse debido al uso del protocolo UDP:

- Paquetes duplicados
- Paquetes perdidos
- Paquetes fuera de orden

La clase Reliability implementa un control de fiabilidad parcial del protocolo UDP, esto es, en el caso de recibir paquetes fuera de orden no se contempla la reordenación de los mismos, ya que supondría tener que almacenar los paquetes hasta recibir aquellos que se habían perdido o llegado tardíamente, lo que generaría un excesivo uso de memoria.

Para mantener éste control en los paquetes recibidos, se añade a los mismos una cabecera que contiene dos parámetros básicos: un número de secuencia local y un número de secuencia remoto.

Su funcionamiento se puede resumir de la siguiente manera:

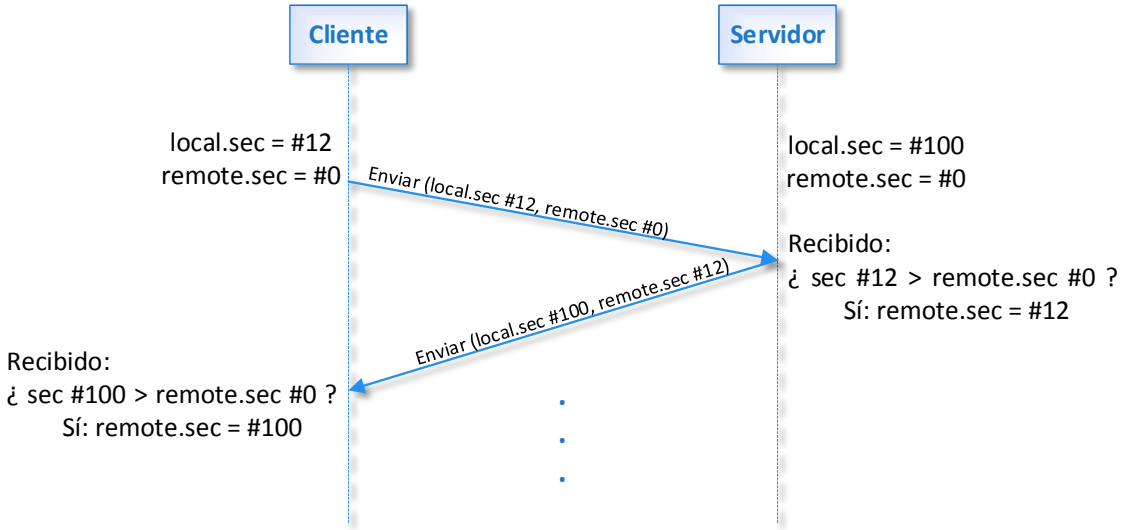

Figura 5.5. Sistema de fiabilidad de UDP

En éste ejemplo, el cliente genera inicialmente el número de secuencia #12: si envía un paquete, la cabecera del mismo contendrá el número de secuencia #12.

El equipo remoto (el servidor en éste caso), recibirá el paquete del cliente y comparará el número de secuencia #12 con el anterior número de secuencia recibido desde el cliente.

La comparación se realiza con el método *CheckAndUpdateSequence()* de la clase Reliability que devuelve *true* si se acepta el paquete cuyo número de secuencia se le pasa como parámetro o *false* si se debe descartar por ser un paquete duplicado o fuera de orden.

Por otra parte, el equipo remoto genera inicialmente el número de secuencia #100: si envía un paquete, la cabecera del mismo contendrá el número de secuencia #100 y además, incluirá el número de secuencia del último paquete recibido desde el cliente, es decir, el #12. Con esto, el equipo remoto estará reconociendo al cliente que ha recibido el paquete con número de secuencia #12, sirviendo éste de acuse de recibo (ACK).

Éste ciclo se repetirá con cada paquete que se envíe desde el cliente hacia el servidor y/o viceversa.

En el momento en que cualquiera de los dos extremos de la comunicación detecte que el tiempo transcurrido supera un timeout determinado, se procederá al reenvío del paquete actual y se volverá a esperar un ACK para el mismo.

Otras consideraciones

Es importante destacar que el sistema de fiabilidad para UDP desarrollado mantiene el control de pérdida y reenvío solamente para el tipo de paquetes de datos. Se debe tener en cuenta la declaración de la constante 'CLI_ACK_TIMEOUT', ya que determina cuánto tiempo habrá que esperar la llegada del acuse de recibo (ACK) para cada paquete enviado.

Éste parámetro determinará también, el tiempo máximo que puede transcurrir entre una llamada a *SendData()* y *RecvData()*. Por ejemplo, si 'CLI_ACK_TIMEOUT = 10000' (en milisegundos), significa que pueden pasar 10 segundos como máximo entre el envío de datos con *SendData()* y la recepción de datos con *RecvData()* o viceversa.

Además, por motivos de simplicidad y limitaciones de memoria en microcontroladores, se ha decidido utilizar un sólo paquete ACK por cada paquete de datos enviado.

Sin embargo, la clase Reliability ha sido desarrollada para ofrecer la posibilidad de que un conjunto de paquetes enviados puedan ser reconocidos todos a la vez con la recepción de un único paquete ACK (revisando la máscara de bits *ackBitfield*).

Esto implicaría tener que almacenar temporalmente dicho conjunto de paquetes, con lo que el consumo de memoria sería mucho mayor, imposibilitando el uso de ésta característica en algunos tipos de microcontroladores.

5.2.4. Control de versiones de firmware

Con el fin de que la placa microcontroladora Arduino pueda actualizar automáticamente la aplicación que se ejecuta sobre la misma desde el momento de la puesta en marcha del dispositivo, se requiere la reprogramación del gestor de arranque o bootloader que permita dicha característica.

El hardware Arduino viene de fábrica con un bootloader, el cual se desea substituir por otro que soporte algún sistema de descarga de datos a través de la red local o Internet.

Para simplificar dicha tarea, se ha tomado como base una implementación ya existente de un bootloader que permite la actualización de la memoria flash en procesadores AVR a partir de un archivo de firmware y que utiliza el protocolo TFTP para la descarga del mismo.

El proyecto se conoce como AVR-Etherboot y sin entrar en profundidad en los detalles de la implementación del mismo, se procederá a explicar la implementación del código desarrollado para realizar el control de versiones de firmware.

El código principal del bootloader está implementado en el archivo *etherflash.c*, en donde se han añadido algunos archivos de cabecera y variables globales:

```
#include  <avr/pgmspace.h>
#include  <string.h>
#include  <stdlib.h>
uint8_t   saveToFlash = 0;
uint8_t   updateRequired = 0;
uint32_t  signatureOffset = 0;
uint32_t  signatureLocation = _VECTORS_SIZE;
```

La ejecución comienza en la función principal *main()* con la inicialización de variables y registros.

Posteriormente se llama a la función *BootLoaderMain()*, en la que se ha añadido el siguiente código:

```
uint32_t  serialInFlash = pgm_read_dword( signatureLocation );
if ( serialInFlash == 0xFFFFFFFF )
        saveToFlash = 1;
```

En este punto se lee el contenido de la dirección *signatureLocation* de memoria flash (que es donde se almacena el valor de versión o firma del firmware) y si es igual al valor 0xFFFFFFFF, se activará el flag *saveToFlash*.

Ésta es una forma de forzar la actualización del firmware sin tener que verificar si el que está ya presente en la memoria flash es anterior o posterior al que se descargue desde el servidor TFTP.

A continuación, se lanza una petición de descarga del archivo con nombre *firmware.hex* al servidor TFTP, llamando a la función *sendTFTPrequest()*.

Si el servidor TFTP no pudiera servir el archivo solicitado por no existir éste o no se obtiene respuesta del mismo por algún problema de conexión en la red, se volverá a reintentar varias veces la solicitud del archivo, con un retardo entre reintentos definido en la constante 'TFTP_TIMEOUT' presente en el archivo *etherflash.h*. De lo contrario, si el servidor TFTP recibe la petición y puede servir el archivo solicitado, entonces comenzará a enviarlo en bloques de 512 bytes.

Dado que la variable global *signatureLocation* apunta a la dirección de la memoria flash '_VECTORS_SIZE' (la última posición de memoria del vector de interrupciones) que suele localizarse en los primeros bytes de la misma, hará falta descargar solamente uno o dos paquetes de 512 bytes para poder comparar la versión del firmware actual con la del que se desea descargar.

De esta manera, cuando el servidor TFTP responda, cada bloque de 512 bytes recibido disparará una llamada *callback* a la función *tftp_get()*, en la cual se ha añadido el siguiente código para analizar cada línea (en formato Intel HEX) del bloque recibido:

```
if ( saveToFlash ) {
        processLineBuffer(sock.lineBufferIdx);
        sock.lineBufferIdx = 0;
}

else {
        if ( checkSignatureInLine( lineBuffer, sock.lineBufferIdx, signatureLocation ) ) {

                if ( updateRequired ) {

                        // Se ha verificado la firma, retornamos y esperamos reintento de
                        // transmisión de archivo para entrar en 'saveToFlash' y guardar los
                        // bloques que recibimos en la flash.
                        saveToFlash = 1;
                        sock.BlockNumber = 0;
                        sock.Bufferfill = 0;
                        sock.lineBufferIdx = 0;
                        return;

                }

                else {
                        // Enviamos mensaje de cancelación de transmisión de archivo y
                        // saltamos directamente al programa presente en la flash.
                        uint8_t *udpSendBuffer = ethernetbuffer + (ETH_HDR_LEN +
                                                                   IP_HDR_LEN +
                                                                   UDP_HDR_LEN);
                        char firmMsg[50];
                        char firmVersion[11];

                        ultoa(pgm_read_dword( signatureLocation ), firmVersion, 10);

                        strcpy(firmMsg, "Error: version de firmware debe ser posterior a ");
                        strcat(firmMsg, firmVersion);

                        udpSendBuffer[0] = 0x00;
                        udpSendBuffer[1] = 0x05;
                        udpSendBuffer[2] = 0x00;
                        udpSendBuffer[3] = 0x05;
                        memcpy(udpSendBuffer + 4, firmMsg, strlen(firmMsg));

                        UDP_SendPacket (4 + strlen(firmMsg), sock.SourcePort,
                                                             sock.DestinationPort,
                                                             sock.DestinationIP);

                        UDP_UnRegisterSocket(sock.SourcePort);

                        jumpToApplication();
                }
        }

        else {
                sock.lineBufferIdx = 0;
        }

}
```

Si el flag *saveToFlash* se encuentra a *true* entonces se analizará la línea, se procesará y convertirán los valores hexadecimales a formato byte y se escribirán en la memoria flash. De lo contrario, si el flag *saveToFlash* se encuentra a *false* se verificará si la firma está presente en alguna línea del bloque descargado, llamando a la función *checkSignatureInLine()*:

```
uint8_t checkSignatureInLine(uint8_t *hexBuffer,
                             uint8_t lineLength,
                             uint32_t signatureAddress) {

    // Ignoramos primer byte ':' y convertimos parejas de caracteres ASCII a binario.
    // Recorremos línea hasta 'lineLength - 3' porque quitamos los últimos 2 bytes de CRC.

    uint8_t  i;
    uint8_t  line[46];
    uint32_t signatureFound = 0;
    for ( i = 0;  i < ((lineLength - 3)/2); i++) {
        line[i] = hexToByte(hexBuffer, (i*2) + 1);
    }

    //Formato Intel HEX (descartando el primer byte ':' y último byte de CRC)
    //line[0] -> Longitud
    //line[1] -> Dirección
    //line[3] -> Tipo de registro
    //line[4] -> Primer byte de datos

    for ( i = 0; i < line[0]; i++ ) {

        if ((signatureOffset >= signatureAddress) &&
            (signatureOffset <= signatureAddress + 3))

            // Los datos comienzan en 'line[i + 4]'.
            signatureFound = (signatureFound << 8) | (line[i + 4]);

        signatureOffset++;

    }

    if ( signatureFound ) {

        if ( checkUpdateRequired( htonl(signatureFound) ) )
            updateRequired = 1;

        return 1;

    }

    return 0;

}
```

Ésta función analiza cada línea en formato hexadecimal pasada como parámetro, la convierte a formato byte y verifica si se ha alcanzado el *offset* del byte donde se almacena la firma (se verificarán 4 bytes ya que la firma es un entero de 32 bits).

Cuando se alcanza la firma, se verificará si ésta es posterior a la del firmware presente en la memoria flash, llamando a la función *checkUpdateRequired()*, que determina finalmente si es necesario actualizar el firmware, en cuyo caso *saveToFlash* se establecerá a *true* y se solicitará nuevamente el reenvío del archivo, que se escribirá directamente en la memoria flash sin verificar nuevamente la firma.

A pesar de tener que solicitar nuevamente el reenvío del archivo, se trata de un método eficiente ya que solamente se verificarán como máximo los dos primeros bloques de 512 bytes.

La función *checkUpdateRequired()* descompone la firma que recibe como parámetro, en un formato de año, mes, y revisión, cuyo código es el siguiente:

```
uint8_t checkUpdateRequired(uint32_t signatureCheck) {

        uint16_t newYear = (signatureCheck / 1000000);
        uint16_t newMonth = ((signatureCheck / 10000) % 100);
        uint16_t newRev = (signatureCheck % 10000);

        uint32_t signatureInFlash = pgm_read_dword( signatureLocation );

        uint16_t prevYear = (signatureInFlash / 1000000);
        uint16_t prevMonth = ((signatureInFlash / 10000) % 100);
        uint16_t prevRev = (signatureInFlash % 10000);


        if ( newYear > prevYear )
                return 1;

        else if ( newYear == prevYear ) {

                if ( newMonth > prevMonth )
                        return 1;

                else if ( newMonth == prevMonth ) {

                        if ( newRev > prevRev )
                                return 1;

                }

        }

        return 0;

}
```

Finalmente, si no se requiere actualizar el firmware, se enviará un mensaje de cancelación de transmisión del archivo al servidor y se saltará directamente al programa presente en la flash con la llamada a la función *jumpToApplication()*.

Es imprescindible tener en cuenta que la configuración estática de IP local, puerta de enlace, IP de servidor TFTP y nombre de archivo a descargar, son parámetros modificables en el archivo *eemem.c*.

Si está habilitada la opción DHCP para obtener los parámetros de conexión automáticamente, entonces se buscará un servidor TFTP en la subred local haciendo una petición *broadcast*, aunque es posible configurar el servidor DHCP para enviar explícitamente la IP de un servidor TFTP y el nombre de un archivo de firmware.

6. Evaluación y Resultados

6.1. Conexión de dispositivos en red

Se han realizado pruebas sobre diversas configuraciones de red para la verificación y validación del sistema desarrollado.

Se plantea como posible escenario una topología de red en la cual los dispositivos Arduino se encuentran distribuidos en redes de área local (LAN), conectados a sus respectivas puertas de enlace o encaminadores (routers), pudiendo de esta forma coexistir con otros dispositivos, ya sean PC o equipos móviles (portátiles, smartphones o tablets, entre otros), visibles entre todos ellos.

En el siguiente esquema y a modo de ejemplo, se detallan los componentes típicos que pudieran intervenir en una comunicación con los dispositivos Arduino:

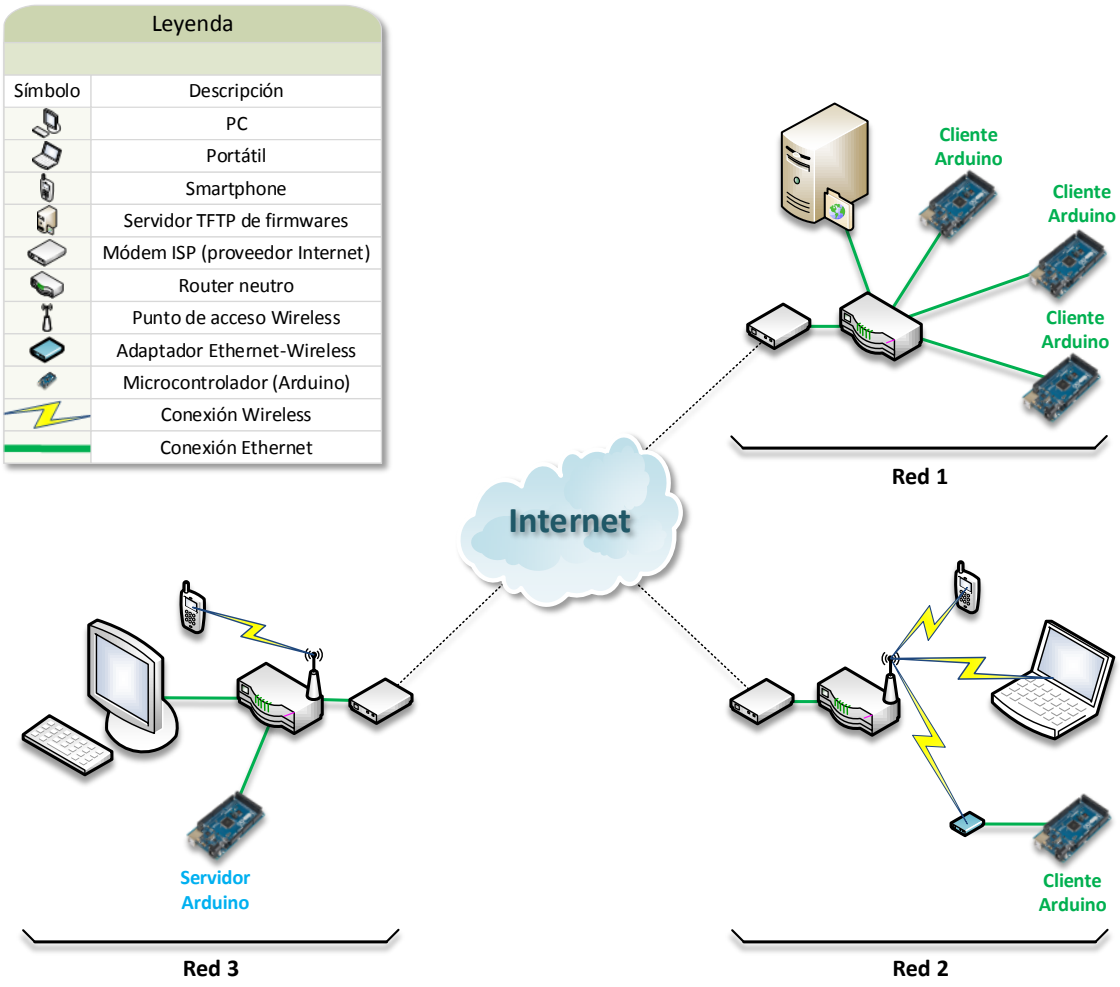

Figura 6.1. Ejemplo de interconexión de dispositivos

La principal ventaja que presenta esta configuración es la de reutilizar la tecnología estándar disponible actualmente para interconectar los dispositivos Arduino, traduciéndose este factor en un mínimo impacto en cambios requeridos en la infraestructura de red.

De esta forma no solo los objetos podrían comunicarse entre sí, sino que dispositivos ya existentes y de uso extendido podrían comunicarse con los objetos, bien para enviarles órdenes (ejecutar comando sobre actuador) o para recibir datos (obtener muestras de datos desde sensores).

Se ha podido verificar la comunicación exitosa entre dispositivos presentes en diferentes redes conectadas a Internet, de forma que un microcontrolador con un sensor de temperatura y humedad presente en la *Red 1*, superado un determinado umbral de temperatura, enviaría un comando que activaría un actuador (relé) conectado a otro microcontrolador en la *Red 2*, pudiéndose comunicar a través del servidor Arduino presente en la *Red 3*.

Así, sólo ha sido necesaria la redirección (NAT) en el router de la *Red 3* del puerto a la escucha del servidor para dar acceso al mismo desde Internet.

La configuración de parámetros de conexión se puede asignar manualmente a cada dispositivo o desde un servidor DHCP (servicio que suele ser ofrecido por los routers actuales, o bien por un servidor dedicado para tal fin).

Para que se pudiera encontrar al servidor a través de un nombre de host o DNS en Internet, se ha utilizado un servicio de soluciones de DNS para direcciones IP dinámicas (No-IP o FreeDNS son algunos ejemplos gratuitos de este servicio).

Respecto a la actualización de firmware desde un servidor TFTP, tanto las pruebas en red local como a través de Internet han sido satisfactorias; cabe mencionar que algunos servidores DHCP tienen la capacidad de ofrecer al cliente la IP y el nombre del archivo del firmware de un servidor TFTP presente en la red local o bien en Internet.

Por lo tanto, en el ejemplo anteriormente planteado, no sólo los dispositivos Arduino presentes en la *Red 1* podrían acceder a su servidor TFTP local, sino que también podrían acceder al mismo los dispositivos Arduino presentes en la *Red 2*, como los de la *Red 3* (se requeriría el redireccionamiento del puerto UDP 69 para TFTP en el router de la *Red 1* para que fuera accesible desde la *Red 2* y desde la *Red 3* a través de Internet).

6.2. Soporte para conexión Wireless

A pesar de que existe la posibilidad de añadir conexión inalámbrica a las placas microcontroladoras Arduino a través de los conocidos *shields* diseñados para tal fin, éstos no supondrían la opción más económica actualmente.

Para poder reutilizar el hardware disponible basado en conexiones de red cableadas y convertirlo en un dispositivo wireless, existen routers wifi que pueden ser establecidos en modo cliente, dando acceso inalámbrico a todos los dispositivos conectados a sus interfaces ethernet. Los inconvenientes de esta solución vienen referidos por un consumo energético y tiempos de inicialización elevados.

Para solventar los mencionados inconvenientes, se propone una solución alternativa basada en un dispositivo portátil con interfaz Ethernet-Wireless, cuyas características se citan a continuación:

- Diversos modos de operación:
  - Router (interfaz ethernet (WAN) adquiere IP ofrecida por ISP).
  - Punto de Acceso Wifi (AP).
  - Modo Cliente (*bridge* o repetidor).

- En modo Cliente, su única interfaz ethernet puede ofrecer conexión a múltiples dispositivos conectados a un switch.

- Tiempo de inicialización reducido al tratarse de un dispositivo embebido que ofrece servicios básicos (no se aprecian diferencias significantes en tiempos de inicialización para configuraciones de IP estática como dinámica por DHCP).

- Bajo consumo energético, aproximadamente 215 mA con la interfaz ethernet inactiva, y unos 245 mA en pleno funcionamiento.

- Dada sus características, se trata de un dispositivo muy económico.

Figura 6.2. Adaptador Ethernet-Wireless BL-MP01

6.3. Simulación de clientes

Debido a las limitaciones de disponibilidad de hardware físico (placas microcontroladoras) para realizar pruebas y dado que una de las principales finalidades era desarrollar un código lo más portable posible a otras plataformas, ha sido necesaria la creación de alguna aplicación que pudiera verificar el funcionamiento del protocolo de comunicación y facilitar las fases de prueba y depuración.

Cada instancia ejecutada de esta aplicación simulará una conexión con el servidor como si se tratase de un dispositivo físico. Por lo tanto para el servidor no habrá distinción alguna entre los mismos.

La interfaz es muy simple y permite la monitorización de otros clientes conectados al mismo servidor. Introduciendo la dirección (nombre de host o IP) y el puerto del servidor, podrá iniciarse una conexión y comenzará la comunicación de datos cuya actividad podrá verificarse mediante los campos *Enviar*, *Recibir* y *Keepalive*.

Para consultar la información de un cliente en concreto, primero se debe obtener una lista de aquellos que estén conectados, como puede observarse aquí:

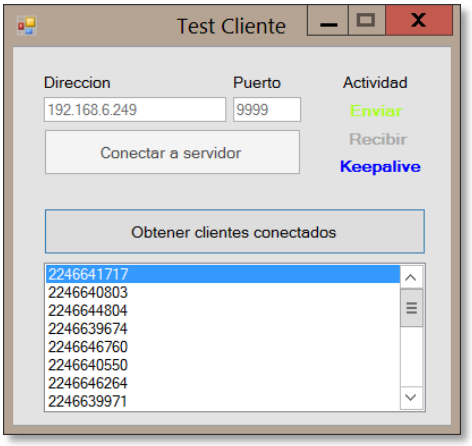

Figura 6.3. Prueba de conexión y monitorización de actividad

Desde la lista de IDs de clientes obtenida se podrá consultar la información relevante de los mismos, la cual se presentará en una nueva ventana y desde la que se podrá interactuar con el cliente remoto enviando comandos disponibles en la lista desplegable inferior, como se observa en la siguiente figura:

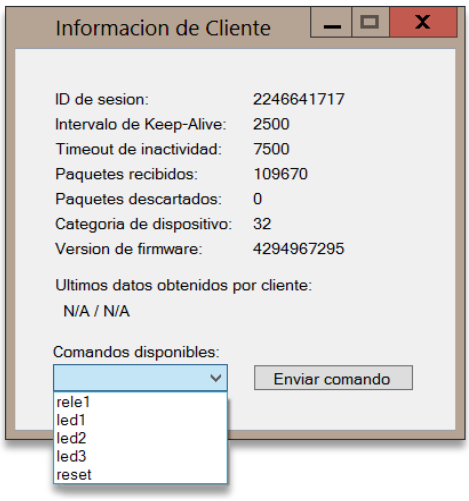

Figura 6.4. Información de cliente

6.4. Versión de servidor para PC

La versión del servidor para PC funciona de igual manera que la versión sobre Arduino, con la salvedad de que han sido necesarios unos mínimos cambios inherentes a la plataforma. Además, ésta versión ofrece un servidor HTTP integrado, ofreciendo una interfaz web sencilla a través de la cual se puede consultar información sobre los clientes conectados.

La implementación actual permite la conexión de cientos de clientes, dado que la capacidad de memoria para PC es muchas veces superior a la de un microcontrolador Arduino. La ejecución del servidor en la línea de comandos se puede llevar a cabo como en el ejemplo de la figura que se presenta a continuación:

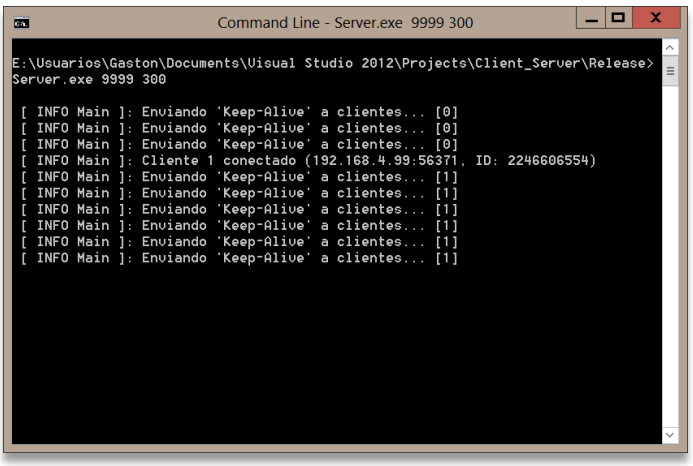

Figura 6.5. Ejecución de aplicación servidor

Como se puede observar, la ejecución de esta instancia del servidor quedaría a la escucha en el puerto UDP 9999 y daría acceso a 300 clientes como máximo, de los cuales se ha conectado uno, mientras se notifica al usuario el momento en que ocurre el envío de mensajes Keep-Alive a todos los clientes.

El acceso a la lista de clientes a través de la interfaz web se realiza con una conexión HTTP al puerto TCP 8080, de la siguiente forma:

Figura 6.6. Acceso a lista de clientes vía web

Y la consulta sobre el cliente devolvería la siguiente información:

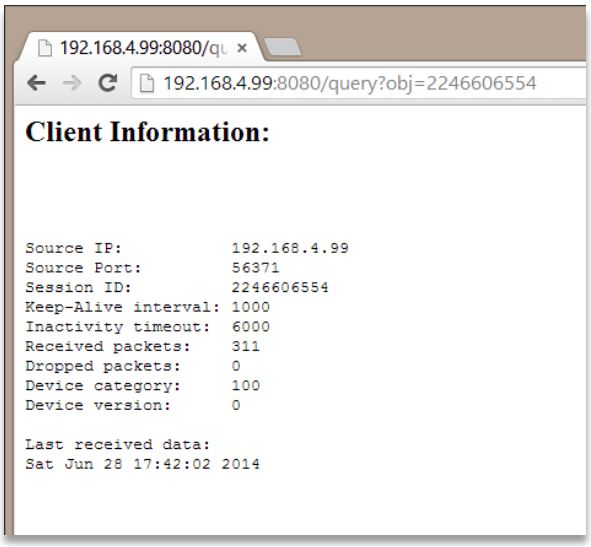

Figura 6.7. Información de cliente vía web

7. Conclusiones

La utilización de la plataforma Arduino para el desarrollo de aplicaciones para Internet de las Cosas es una práctica cada vez más difundida. Frente a otras plataformas representa una opción asequible y de programación lo más sencilla posible.

A pesar de la paulatina aparición de kits de desarrollo para microcontroladores orientados a la programación de aplicaciones para Internet de las Cosas, existe una gran diversidad de productos en el mercado basados en microcontroladores que dificulta la integración de todas sus funcionalidades en una misma API que pueda proporcionar a los desarrolladores una progresiva adaptación para la creación de aplicaciones para Internet de las Cosas.

El presente proyecto ha permitido verificar la viabilidad del uso de Arduino como alternativa económica basada en hardware y software libre, a la vez que se hace posible la extensión de ciertas funcionalidades para facilitar las operaciones típicas en aplicaciones de red, un sistema básico de control de fiabilidad para UDP o una arquitectura cliente/servidor para la interconexión de dispositivos.

Debido a que en la actualidad el campo de Internet de las Cosas está experimentando una rápida y progresiva expansión, cabría realizar un estudio más detallado a medio plazo de otras dificultades que se nos presentan para una implementación eficaz sobre la plataforma Arduino:

- Añadir seguridad en la comunicación (p. ej. protocolo DTLS).
- Mejorar la interacción inteligente entre objetos.
- Flexibilizar intercambio de información mediante la integración de tecnologías Web (REST, XML, JSON).
- Desarrollar aplicaciones para dispositivos móviles para la interacción con objetos inteligentes.
- Integración con protocolo de alto nivel para Internet de las Cosas (CoAP).

8. Anexos

8.1. Anexo I: Hardware AVR compatible: WG2001.NET

Otra interesante alternativa al hardware Arduino es, entre otros, la placa microcontroladora WG2001.NET basada en un microcontrolador AVR Atmega128(L) y cuya utilización en el presente proyecto ha sido muy significativa dado que se integran leds, relés (actuadores) y el propio chip ENC28J60, todo en una misma placa.

A partir del código para el cliente Arduino, la portabilidad del mismo a esta placa ha sido casi directa.

No obstante, dado que el entorno de desarrollo de Arduino está expresamente preparado para soportar la diversidad de placas de la misma plataforma (también Visual Studio con los plugins mencionados con anterioridad), no puede ser utilizado para la placa WG2001.NET y por lo tanto, no dando soporte a su microcontrolador AVR Atmega128(L). Por este motivo ha sido necesario utilizar el entorno de desarrollo Atmel Studio.

Este entorno de desarrollo es similar a Visual Studio y ha facilitado enormemente la adaptación del código desarrollado inicialmente para Arduino.

A diferencia de las placas Arduino, que utilizan el puerto USB para programar o subir el código compilado, la placa WG2001.NET no dispone de tal puerto y sólo puede programarse con un programador ICSP.

Para facilitar aun más el proceso de programación y no depender del programador ICSP para actualizar el programa compilado en la memoria flash, se ha actualizado el bootloader original y substituido por AVR-Etherboot, el cual aprovecha la capacidad de descargar desde un servidor TFTP el firmware más reciente y actualizar la memoria flash.

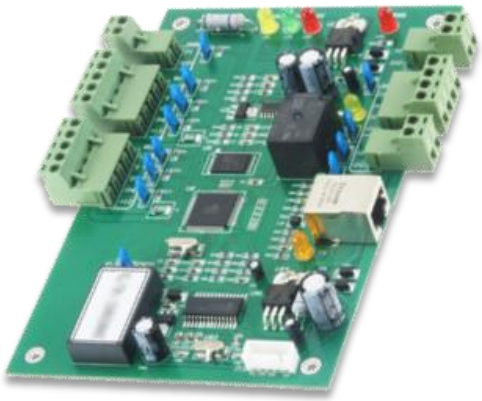

Figura 8.1. Placa WG2001.NET

8.2. Anexo II: Programación de bootloader en Arduino

La programación del bootloader se puede realizar mediante varias aplicaciones que puedan acceder al programador ICSP. Uno de los más utilizados es el conocido como USBasp, con interfaz USB-ISP.

Figura 8.2. Programador USBasp

Una vez conectado, tanto desde el entorno de desarrollo de Arduino como desde Visual Studio, se puede programar el bootloader adecuado para la placa seleccionada. Este es el bootloader por defecto (el que viene instalado de fábrica) y suele encontrarse en la ruta "\hardware\arduino\bootloaders" de la distribución de software de Arduino.

Si lo que se desea es programar otro bootloader no original, entonces la herramienta a utilizar más extendida es *avrdude*.

A continuación se pueden observar las opciones de su ejecución en la línea de comandos:

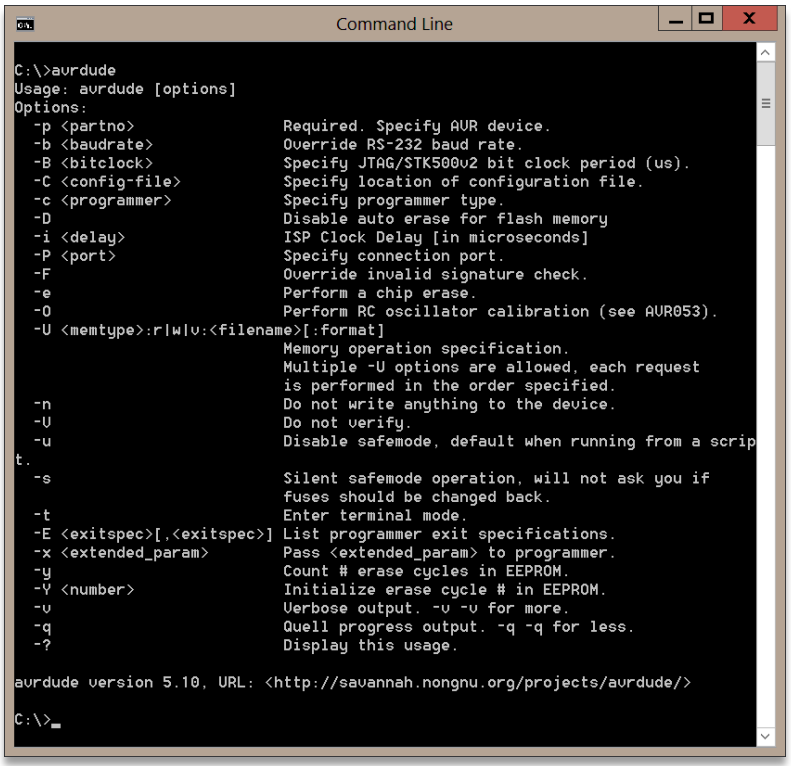

Figura 8.3. Argumentos en línea de comandos de *avrdude*

Para proceder a la programación del microcontrolador, es importante conocer la configuración en concreto de sus *fuses* o fusibles.

Se trata de una memoria de la que disponen los microcontroladores AVR para guardar configuraciones específicas que definen su comportamiento: fuente de reloj, velocidad de reloj, tamaño de la memoria dedicada al bootloader, entre muchos otros.

En el datasheet se describe en detalle la función de cada fuse y el significado de cada bit que lo compone.

Básicamente los pasos a seguir son los siguientes:

a. Desbloquear la sección del gestor de arranque en el chip
b. Fijar los fuses en el chip
c. Subir el código del gestor de arranque al chip
d. Bloquear la sección del gestor de arranque en el chip

Las órdenes que deben ejecutarse en la línea de comandos para realizar los pasos anteriores son las siguientes:

(Ejemplo para microcontrolador Atmega2560):

```
avrdude -v -p atmega2560 -c usbasp -V -e -U lfuse:w:0xFF:m -U hfuse:w:0xD8:m
        -U efuse:w:0xFD:m

avrdude -v -p atmega2560 -c usbasp -V -D -U flash:w:Nuevo_Bootloader.hex

(Opcional si algunos datos deben ser accesibles desde la memoria EEPROM)
avrdude -v -p atmega2560 -c usbasp -V -D -U eeprom:w:Nuevo_Bootloader.eep

avrdude -v -p atmega2560 -c usbasp -V -U lock:w:0x3F:m
```

8.3. Anexo III: Configuración de Visual Studio para Arduino

Para utilizar Visual Studio con soporte para Arduino, se debe instalar Visual Micro, disponible en http://www.visualmicro.com.

Una vez descargado el instalador ejecutable, la instalación es directa y al finalizar esta, Visual Studio se habrá configurado automáticamente.

No obstante, al iniciar Visual Studio tras la instalación de Visual Micro, aparecerá un cuadro de diálogo solicitando la ruta de instalación de la distribución de software de Arduino, como se aprecia en la siguiente figura:

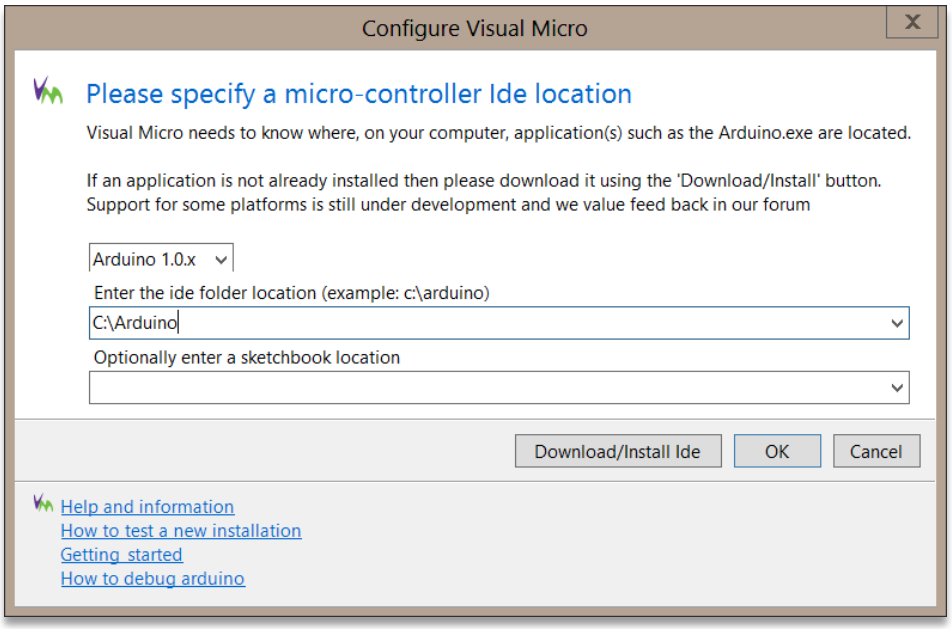

Figura 8.4. Especificación de rutas de instalación de Arduino para Visual Micro

Si posteriormente se realizan cambios en las rutas de instalación de Arduino, deberán actualizarse para Visual Micro también.

Para volver a acceder al cuadro de diálogo y así configurar las rutas nuevamente, se debe hacer desde el menú "Herramientas":

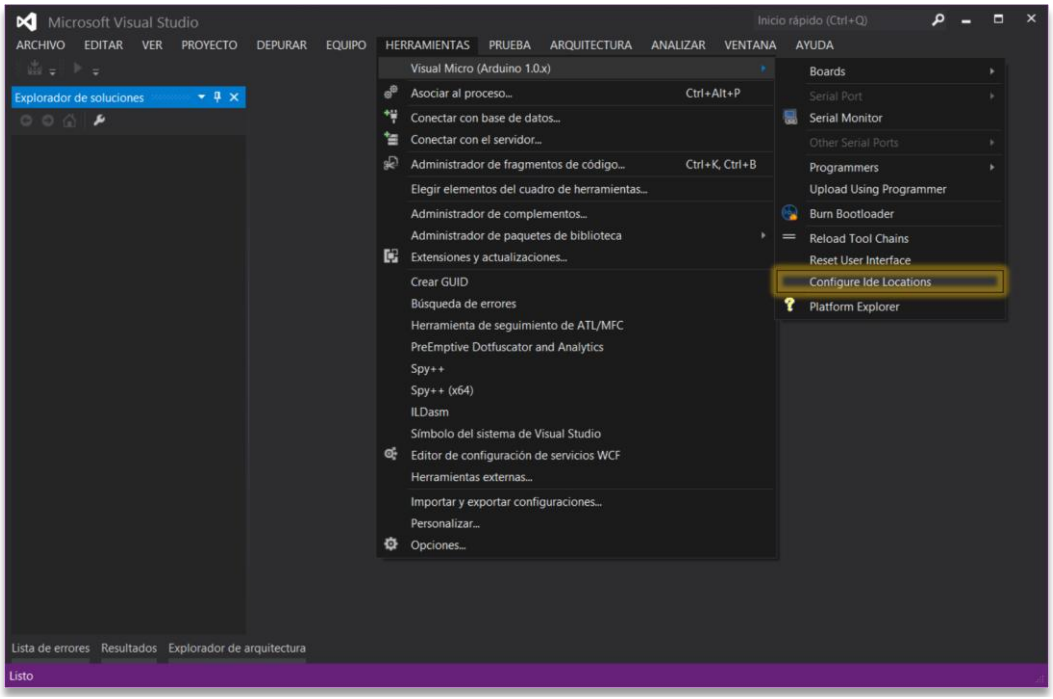

Figura 8.5. Acceso al cuadro de diálogo para configuración de rutas de Visual Micro

8.4. Anexo IV: Configuración de dispositivo wireless

Para dar soporte wireless al módulo de red ENC28J60, tal y como se ha descrito en el capítulo 6.2, se ha utilizado el dispositivo portátil B-LINK BL-MP01.

La configuración básica necesaria para su funcionamiento es la siguiente:

1) Posicionar interruptor lateral en modo *Repeater* (el dispositivo se establecerá en modo cliente para conectarse a otro punto de acceso o AP)

2) Conectar a un puerto USB (como toma de alimentación de 5V)

3) Conectar a través de wifi desde otro equipo:

SSID: B-LINK_<código>
Pass: <vacío>

Notas:

- El código que forma parte del SSID puede variar entre dispositivos.

- El dispositivo asignará una dirección IP mediante DHCP en el rango 192.168.16.100-200 al equipo desde el cual se accede.

4) Acceder a la configuración web a través de http://192.168.16.254

5) Configuraciones básicas:

→ Wireless
  → Security
    → Security Mode: WPA PSK WPA2 PSK
    → WPA Algorithms: TKIP AES

  → Repeater
    → Seleccionar BSSID (dirección MAC) de AP al que se desea conectar en modo cliente
    → Connect
      → Introducir SSID del AP (nombre del AP)
      → Introducir clave para SSID
      → Apply

→ Reiniciar (dispositivo solicitará reinicio reconectando USB)

Con estas configuraciones, el BL-MP01 asignará IPs por DHCP tanto para la interfaz ethernet como para la interfaz wifi. Si el BL-MP01 ha conseguido conectar en modo cliente al AP que da acceso a Internet, entonces todos los dispositivos conectados al mismo tanto por la interfaz ethernet como wifi también tendrán salida a Internet.

9. Glosario

ACK (Acknowledgment): Acuse de recibo o mensaje que el destino en una comunicación envía al origen de ésta para confirmar la recepción de un mensaje.

AP (Access Point): Un punto de acceso inalámbrico es un dispositivo que interconecta dispositivos de comunicación alámbrica para formar una red inalámbrica.

AVR: Familia de microcontroladores RISC del fabricante Atmel.

Avrdude: Herramienta de línea de comandos para la programación de microcontroladores Atmel.

Bridge: Dispositivo de interconexión de redes de ordenadores que opera en la capa 2 (nivel de enlace de datos) del modelo OSI.

Buffer circular: Estructura de datos que utiliza un buffer único o array ordinario y que adopta su nombre por la forma en que se ponen o extraen sus elementos.

Broadcast: Forma de transmisión de información donde un nodo emisor envía información a una multitud de nodos receptores de manera simultánea, sin necesidad de reproducir la misma transmisión nodo por nodo.

CPU (Central Processing Unit): Unidad Central de Procesamiento o procesador, es el componente principal del ordenador y otros dispositivos programables, que interpreta las instrucciones contenidas en los programas y procesa los datos.

Callback: En programación de computadoras, una devolución de llamada o retrollamada es una función "A" que se usa como argumento de otra función "B". Cuando se llama a "B", ésta ejecuta "A". Para conseguirlo, usualmente lo que se pasa a "B" es el puntero a "A".

CoAP (Constrained Application Protocol): Protocolo de software utilizado en dispositivos electrónicos simples el cual les permite comunicarse interactivamente en Internet.

Condición de carrera: Expresión usada en electrónica y en programación cuando el resultado de múltiples procesos depende del orden en que se ejecute. Si los procesos que están en condición de carrera no son correctamente sincronizados, puede producirse un error de corrupción de datos.

Datasheet: Documento que resume el funcionamiento y otras características de un componente (por ejemplo, un componente electrónico) o subsistema (por ejemplo, una fuente de alimentación) con el suficiente detalle para ser utilizado por un ingeniero de diseño y diseñar el componente en un sistema.

DHCP (Dynamic Host Configuration Protocol): Protocolo de red que permite a los clientes de una red IP obtener sus parámetros de configuración automáticamente.

Dirección IP: Etiqueta numérica que identifica, de manera lógica y jerárquica, a una interfaz (elemento de comunicación/conexión) de un dispositivo (habitualmente una computadora) dentro de una red que utilice el protocolo IP.

DNS (Domain Name System): Sistema de nomenclatura jerárquica para computadoras, servicios o cualquier recurso conectado a Internet o a una red privada. Su función más importante, es traducir (resolver) nombres inteligibles para las personas en identificadores binarios asociados con los equipos conectados a la red, con el propósito de poder localizar y direccionar estos equipos mundialmente.

DTLS (Datagram Transport Layer Security): Protocolo que proporciona privacidad en las comunicaciones para protocolos de datagramas.

Ethernet: Familia de tecnologías para redes de área local (LAN).

FIFO (First In, Fisrt Out): Método de organizar y manipular un buffer o pila de datos, donde el elemento más viejo o inferior en la pila es procesado en primer lugar.

Firmware: Bloque de instrucciones máquina para propósitos específicos, grabado en una memoria, normalmente de lectura/escritura (ROM, EEPROM, flash, etc.), que establece la lógica de más bajo nivel que controla los circuitos electrónicos de un dispositivo de cualquier tipo.

Fuse: Área de memoria (suelen ser 4 bytes) en microprocesadores AVR disponibles para su programación que contienen información referente a la configuración del chip.

Handshake*:* Proceso automatizado de negociación que establece de forma dinámica los parámetros de un canal de comunicaciones establecido entre dos entidades antes de que comience la comunicación normal por el canal. De ello se desprende la creación física del canal y precede a la transferencia de información normal.

HTTP (Hypertext Transfer Protocol): Protocolo usado en cada transacción de la World Wide Web.

ICSP (In Circuit Serial Programming): Método de programación directa de microcontroladores AVR y otros.

ID (Identifier): Nombre que identifica un objeto único o una única clase de objetos.

IETF (Internet Engineering Task Force): Organización internacional abierta de normalización, que tiene como objetivos el contribuir a la ingeniería de Internet, actuando en diversas áreas, como transporte, encaminamiento, seguridad.

Interrupción: Señal recibida por el procesador de un ordenador, indicando que debe "interrumpir" el curso de ejecución actual y pasar a ejecutar código específico para tratar esta situación.

IP (Internet Protocol): Protocolo de comunicación de datos digitales clasificado funcionalmente en la Capa de Red según el modelo internacional OSI.

IPv4: Cuarta versión del protocolo Internet Protocol (IP), y la primera en ser implementada a gran escala.

IPv6: Versión del protocolo Internet Protocol (IP), definida en el RFC 2460 y diseñada para reemplazar a Internet Protocol version 4.

ISR (Interrupt Service Routine): Conocido como Manejador de Interrupción o Rutina de Servicio de Interrupción, es una retrollamada (callback) presente en el firmware de microcontroladores, sistemas operativos y drivers de dispositivos, cuya ejecución es disparada por la recepción de una interrupción.

Intel HEX: Formato de archivo para la programación de microcontroladores, EPROMs y otros circuitos integrados.

JSON (JavaScript Object Notation): Formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML.

Keepalive: Mensaje enviado por un dispositivo a otro para comprobar que la línea entre los dos es operativa o para prevenir la pérdida de dicha línea de conexión.

LAN (Local Area Network): Red de área local es una red de computadoras que interconecta computadoras dentro de un área limitada como una casa, una escuela, un laboratorio de computadoras u oficinas, utilizando medios de red.

LCD (Liquid Crystal Display): Una pantalla de cristal líquido es una pantalla delgada y plana formada por un número de píxeles en color o monocromos colocados delante de una fuente de luz o reflectora. A menudo se utiliza en dispositivos electrónicos de pilas, ya que utiliza cantidades muy pequeñas de energía eléctrica.

Microcontrolador: Circuito integrado programable, capaz de ejecutar las órdenes grabadas en su memoria.

NAT (Network Address Translation): La Traducción de Dirección de Red es un mecanismo utilizado por routers IP para intercambiar paquetes entre dos redes que asignan mutuamente direcciones incompatibles.

Offset: Término que indica la distancia (desplazamiento) desde el inicio de un objeto hasta un punto o elemento dado, presumiblemente dentro del mismo objeto. Suele utilizarse en referencia a arrays u otras estructuras de datos.

Polling: Operación de consulta constante, generalmente hacia un dispositivo de hardware, para crear una actividad sincrónica sin el uso de interrupciones, aunque también puede suceder lo mismo para recursos de software.

Preemptivo (Preemptive multitasking): Multitarea apropiativa es una manera en que los sistemas operativos pueden proveer multitarea, es decir, la posibilidad de ejecutar múltiples procesos al mismo tiempo. Al proceso actual, el sistema le asigna un intervalo de tiempo para ejecutarse; una vez acabado el tiempo, el proceso queda pausado y se destina el siguiente intervalo de tiempo a un proceso distinto.

REST (Representational State Transfer): La Transferencia de Estado Representacional es una técnica de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.

RFC (Request for Comments): Conjunto de publicaciones del Internet Engineering Task Force (IETF) que describen diversos aspectos del funcionamiento de Internet y otras redes de computadoras, como protocolos, procedimientos, etc., o comentarios e ideas sobre estos.

RISC (Reduced Instruction Set Computer): Tipo de diseño de CPU generalmente utilizado en microprocesadores o microcontroladores.

RJ45 (registered jack 45): Interfaz física comúnmente usada para conectar redes de cableado estructurado (categorías 4, 5, 5e, 6 y 6a).

RTOS (Real Time Operating System): Un sistema operativo de tiempo real es un sistema operativo que ha sido desarrollado para aplicaciones de tiempo real. Como tal, se le exige corrección en sus respuestas bajo ciertas restricciones de tiempo.

Round-Robin: Método para seleccionar todos los elementos en un grupo de manera equitativa y en un orden racional, normalmente comenzando por el primer elemento de la lista hasta llegar al último y empezando de nuevo desde el primer elemento.

Router: Un enrutador o encaminador es un dispositivo que proporciona conectividad a nivel de red o nivel tres en el modelo OSI. Su función principal consiste en enviar o encaminar paquetes de datos de una red a otra, es decir, interconectar subredes.

SPI (Serial Peripheral Interface): Estándar de comunicaciones, usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos.

Scheduler: El planificador es un componente funcional de los sistemas operativos multitarea y multiproceso, y es esencial en los sistemas operativos de tiempo real. Su función consiste en repartir el tiempo disponible de un microprocesador entre todos los procesos que están disponibles para su ejecución.

Sección crítica: Porción de código de un programa de computador en la cual se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un proceso o hilo en ejecución.

Serialización: Proceso de codificación de un objeto en un medio de almacenamiento (como puede ser un archivo, o un buffer de memoria) con el fin de transmitirlo a través de una conexión en red como una serie de bytes o en un formato humanamente más legible como XML o JSON, entre otros.

Socket (sockets BSD): Concepto abstracto por el cual dos programas (posiblemente situados en computadoras distintas) pueden intercambiar cualquier flujo de datos, generalmente de manera fiable y ordenada. El término socket es también usado como el nombre de una interfaz de programación de aplicaciones (API) para la familia de protocolos de Internet TCP/IP, provista usualmente por el sistema operativo.

TCP (Transmission Control Protocol): Se refiere a uno de los protocolos fundamentales de Internet y provee una entrega fiable, ordenada y con control de errores de octetos entre programas corriendo en computadoras conectadas en una red de área local, intranet o Internet.

TFTP (Trivial File Transfer Protocol): Protocolo de transferencia muy simple semejante a una versión básica de FTP. TFTP a menudo se utiliza para transferir pequeños archivos entre ordenadores en una red.

Timeout: Período de tiempo que será permitido que transcurra en un sistema antes de que un evento especificado tenga lugar, a menos que otro evento ocurra primero. En cualquier caso, el período se termina cuando cualquiera de los dos tiene lugar.

UDP (User Datagram Protocol): Protocolo del nivel de transporte basado en el intercambio de datagramas (Encapsulado de capa 4 Modelo OSI).

USB (Universal Serial Bus): Estándar industrial que define los cables, conectores y protocolos usados en un bus para conectar, comunicar y proveer de alimentación eléctrica entre ordenadores y periféricos y dispositivos electrónicos.

WOL (Wake on LAN): Estándar de redes de computadoras Ethernet que permite encender remotamente computadoras apagadas.

Web: World Wide Web (WWW) o Red informática mundial comúnmente conocida como la web, es un sistema de distribución de documentos de hipertexto o hipermedios interconectados y accesibles vía Internet.

Wifi: Mecanismo de conexión de dispositivos electrónicos de forma inalámbrica.

Wireless: La comunicación inalámbrica o sin cables es aquella en la que la comunicación (emisor/receptor) no se encuentra unida por un medio de propagación

físico, sino que se utiliza la modulación de ondas electromagnéticas a través del espacio.

XML (eXtensible Markup Language): Lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible.

10. Referencias

Arduino Mega 2560:
http://arduino.cc/en/Main/arduinoBoardMega2560

Sensores:
http://es.wikipedia.org/wiki/Sensor

Conectividad - módulo de red ENC28J60:
http://ww1.microchip.com/downloads/en/DeviceDoc/39662e.pdf

Programación sobre plataforma Arduino:
http://es.wikipedia.org/wiki/Arduino
http://arduino.cc/es/Guide/Environment#libraries
http://playground.arduino.cc/ArduinoNotebookTraduccion/Structure

Entorno de desarrollo:
http://arduino.cc/es/Guide/Environment

Visual Studio como entorno de desarrollo:
http://www.visualmicro.com

Sistema operativo de tiempo real ChibiOS/RT:
http://www.chibios.org
http://www.chibios.org/dokuwiki/doku.php?id=chibios:documents

Métodos de E/S: Interrupciones y Polling:
http://web.engr.oregonstate.edu/~traylor/ece473/lectures/interrupts.pdf

Sincronización entre interrupciones y tareas:
http://forum.arduino.cc/index.php/topic,45239.0.html
http://www.netrino.com/Embedded-Systems/How-To/RTOS-Preemption-Multitasking
http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts#interrupts_handling

El protocolo UDP:
http://en.wikipedia.org/wiki/User_Datagram_Protocol

El protocolo TFTP:
http://en.wikipedia.org/wiki/Trivial_File_Transfer_Protocol

Otros protocolos basados en UDP para Internet de las Cosas:
http://postscapes.com/internet-of-things-protocols
http://www.embedded.com/electronics-blogs/cole-bin/4229531/UDP---the-embedded-wireless--Internet-of-Things-

Glosario:
http://es.wikipedia.org