Resumen

El presente proyecto describe cómo utilizar la pila de comunicaciones “Stack TCP/IP” de Microchip con microcontroladores PIC18 de 8 bits, habitualmente utilizados en entornos académicos.

Debido a la limitación de memoria RAM del microcontrolador escogido, el PIC18LF4680, se han analizado y evaluado aplicaciones muy concretas del uso del stack.

Para realizar dicha evaluación se propone la construcción de tres aplicaciones que sirven para la captura de temperaturas en tiempo real utilizando un sensor DS18B20 de Maxim Integrated. En cada una de estas aplicaciones se analiza, a diferentes niveles, el uso de los protocolos de comunicación que contiene el stack.

La primera aplicación consiste en la implementación de un servidor web dentro de una red ad-hoc con un PC. En ésta se analiza la gestión del protocolo de comunicación HTTP del que hace uso el PIC18. También se muestra la utilización de variables dinámicas dentro de la web almacenada, para la visualización de la temperatura captada por el sensor.

La segunda aplicación consiste en la creación de una red de PCs y dos PIC18 que se comunican a través del protocolo UDP, ampliamente utilizado en transmisiones de datos que no requieren del establecimiento de una conexión. La comunicación a través del PC se realiza con un script de Python. Éste actúa como servidor a la escucha y los PICs como clientes que envían la temperatura a toda la red. El script de Python muestra por la pantalla del PC la temperatura enviada. Se analizan los pasos para la comunicación con protocolo UDP del que hace uso el stack.

La tercera aplicación consiste en el montaje de un servidor web en un primer PIC18, que además de mostrar la temperatura en el navegador de un PC conectado en su misma red, pueda mostrar la temperatura de otro sensor (segundo PIC18). Para ello se constituye una estructura cliente/servidor entre los dos PIC18/sensores DS18B20. En este caso el protocolo de comunicación entre cliente / servidor es TCP/IP. El PC controla con un método GET el refresco de datos de temperatura mostrados en el navegador. De forma análoga se analizan los pasos para la comunicación con protocolo TCP/IP del que hace uso el stack.

Sumario

1. Glosario

MCU: Microcontrolador

PIC: Microcontroladores fabricados por Microchip

OSI: Open System Interconnection

Stack: Pila, estructura de datos.

Socket: Puerta de comunicaciones

RFC: Request for Comments, publicaciones del IETF.

IETF: Internet Engineering Task Force

ARP: Address Resolution Protocol

MAC: Media Access Control

IP: Internet Protocol

UDP: User Datagram Protocol

TCP: Transport Control Protocol

2. Introducción

En el mundo globalizado e hipercomunicado en el que vivimos, los protocolos de comunicación juegan un papel básico en las operaciones diarias de muchas personas, empresas e instituciones. Sobre estas reglas de comunicación establecemos aplicaciones que permiten conectar e interactuar a dispositivos situados a miles de quilómetros unos de otros.

Cuando se habla de protocolos de comunicación, se habla de reglas estandarizadas para el establecimiento de comunicación entre diferentes dispositivos. Es importante recalcar que no existe un único protocolo que englobe todos los niveles de la comunicación, sino un conjunto de ellos que procesan la información a distintos niveles.

Es la International Organization for Standardization (ISO) la que establece 7 niveles o capas de comunicación a través del modelo Open System Interconnection (OSI) en su norma ISO/IEC 7498-1 [1].

Cada una de las capas del modelo OSI hace referencia a una funcionalidad específica y normalizada de la red, lo que ha permitido el desarrollo de distintos protocolos intercambiables a distintos niveles de comunicación.

La gestión de la información entre los diversos niveles se realiza *encapsulando* la información en paquetes, a los cuales se les añade la información del protocolo necesaria para ser enviado a través de una red [2].

A partir del modelo de referencia OSI se desarrolló el modelo de referencia TCP/IP (Protocolo de control de transmisión/Protocolo Internet), que hizo posible el desarrollo de Internet y en el que se ha centrado este proyecto.

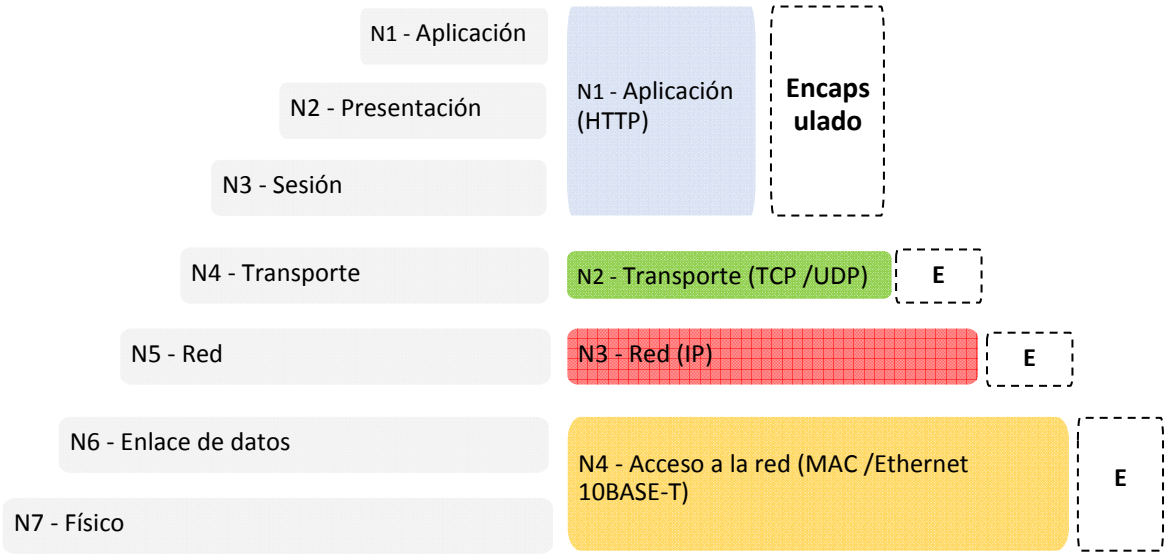


Fig. 2.1 Pirámide OSI y Modelo TCP/IP Fuente: Propia.

Como implementar el código para cada uno de los niveles de comunicación, cada vez que hubiera un requerimiento de comunicación entre microcontroladores, supondría muchas horas de desarrollo, surge la necesidad de crear un conjunto de instrucciones que se encarguen de gestionar la transmisión de la información entre los distintos niveles. Este conjunto de instrucciones de código, basado en la pirámide OSI bajo el modelo TCP/IP de comunicación e implementado por Microchip, es lo que a partir de ahora en el proyecto denominaremos pila o *stack* de comunicaciones [3].

2.1. Objetivos del proyecto

Este proyecto pretende servir de base didáctica para alumnos del Grado y del Máster Universitario en Tecnologías Industriales en la compresión del modelo TCP/IP, a la vez que se evalúa el correcto funcionamiento, la idoneidad y las posibilidades de uso del stack de comunicaciones desarrollado por Microchip con microcontroladores (MCU) de 8 bits con poca capacidad de memoria (tanto RAM como ROM), como los utilizados durante las prácticas del Grado y del Máster de Ingeniería Industrial.

La metodología seguida ha consistido en evaluar la capacidad del microcontrolador PIC18LF4680 para ejecutar una aplicación de captura de temperaturas en tiempo real, a la

vez que se ejecuta el stack de comunicaciones para la transmisión de estos datos utilizando los protocolos de transporte UDP y TCP, así como su uso para la creación de una aplicación HTTP.

Para la demostración didáctica de las capacidades del stack se han creado tres aplicaciones (máquinas) que van mostrando con un nivel de complejidad conceptual cada vez mayor el uso de distintos protocolos y aplicaciones del stack a diferentes niveles de comunicación del modelo TCP/IP.

A medida que se ejecute el código de cada máquina y se vaya accediendo a distintos protocolos de comunicación, estos son explicados a nivel conceptual. Además se analiza cómo se ha implementado dicho protocolo en el stack.

Al tratarse de un proyecto con una importante voluntad didáctica, todos los proyectos, aplicaciones, archivos de captura de paquetes y scripts utuilizados se encuentran incluidos dentro de sus carpetas correspondientes para facilitar la repetitividad de los resultados a cualquier alumno interesado.

2.2. Alcance del proyecto

A continuación, encontramos una breve descripción de los objetivos planteados en cada una de las tres aplicaciones (máquinas).

MAQ1-Servidor Web

En la primera aplicación se muestra en una web la temperatura captada por un sensor situado en una placa de desarrollo. Se observarán las siguientes características del stack:

- Utilizar el microcontrolador como servidor Web, creación de una red Ad-hoc con un PC y visualización de la temperatura en éste.
- Implementación de una página web en la memoria del microcontrolador que utilice *Variable Dinámicas* (HTTP) para el envío de información al navegador.
- Análisis de la implementación que se hace en el stack del protocolo de comunicación a nivel de aplicación (HTTP) y uso del software Wireshark para la visualización de paquetes de datos.

MAQ2-Servidor UDP

En la segunda aplicación se implementa un servicio básico de cliente-servidor UDP para mostrar la temperatura de dos placas en el terminal de un PC:

- Utilizar dos microcontroladores como clientes UDP, los cuales envíen a cualquier PC situado en su misma red una actualización de la temperatura de ambas unidades.
- Creación de un script en Python que sirva como servidor UDP en el PC para la visualización de las temperaturas.
- Análisis del protocolo UDP y su implementación en el stack de Microchip. Visualización de paquetes de datos con el software Wireshark.

MAQ3-Servidor Web con Servidor TCP

En la tercera aplicación se implementa una web a través de cual el usuario solicita el envío de temperaturas de dos sensores situados en dos placas. Además, se implementa un servicio cliente-servidor TCP para la comunicación entre las dos placas. Se trata mostrar en una única web la temperatura de dos sensores situados en las dos placas y gobernar la actualización de datos de ambas placas desde la propia web. La máquina dispone de las siguientes características:

- Utilizar un microcontrolador como servidor Web dentro de una red con un PC y un switch por el que se interconectarán el PC y ambas placas (MCU).
- Control de actualización de la temperatura desde el ordenador de ambas placas (MCU) en una única web.
- Implementación de una página web en la memoria del MCU que utilice *Variables Dinámicas* (HTTP) para la visualización de la web y un método *GET* (HTTP) para gobernar la actualización de datos.
- Creación de un servidor TCP/IP en el segundo MCU, al que accederá el primer MCU mediante un cliente TCP/IP para obtener la temperatura del segundo sensor y guardarlo en el servidor web.
- Análisis del protocolo TCP e implementación en el stack de Microchip. Visualización de paquetes de datos con el software WireShark.

2.3. Antecedentes

El stack TCP/IP de Microchip utilizado en este proyecto suele ser utilizado en proyectos que incluyen microcontroladores de gran potencia de cálculo y capacidad de memoria, como por ejemplo:

- MCUs de la familia PIC24 de 16 bits, como el PIC24FJ64GA004
- MCUs de la familia PIC32 de 32 bits, como el PIC32MX460F512L

Para estos MCUs el stack dispone de funciones de *autoconfiguración* facilitando su uso en gran medida. Se debe señalar que aunque el stack también dispone de funciones de *autoconfiguración* para PIC18 como las observadas en el código mostrado a continuación, éstas hacen referencia a MCUs que, o bien disponen de controladores de Ethernet integrados o utilizan placas comerciales o disponen de encapsulados distintos al utilizado.

```
#if defined(__18F8722) && !defined(HI_TECH_C)
    // PICDEM HPC Explorer or PIC18 Explorer board
    hola
    #pragma config OSC=HSPLL, FCMEN=OFF, IESO=OFF, PWRT=OFF, WDT=OFF, LVP=OFF
#elif defined(_18F8722) // HI-TECH PICC-18 compiler
    // PICDEM HPC Explorer or PIC18 Explorer board with HI-TECH PICC-18 compiler
    __CONFIG(1, HSPLL);
    __CONFIG(2, WDTDIS);
    __CONFIG(3, MCLREN);
    __CONFIG(4, XINSTDIS & LVPDIS);
#elif defined(__18F87J10) && !defined(HI_TECH_C)
    // PICDEM HPC Explorer or PIC18 Explorer board
    #pragma config WDTEN=OFF, FOSC2=ON, FOSC=HSPLL
#elif defined(__18F87J11) && !defined(HI_TECH_C)
    // PICDEM HPC Explorer or PIC18 Explorer board
    #pragma config WDTEN=OFF, FOSC=HSPLL
#elif defined(__18F87J50) && !defined(HI_TECH_C)
    // PICDEM HPC Explorer or PIC18 Explorer board
    #pragma config WDTEN=OFF, FOSC=HSPLL, PLLDIV=3, CPUDIV=OSC1
#elif (defined(__18F97J60) || defined(__18F96J65) || defined(__18F96J60)
    || defined(__18F87J60) || defined(__18F86J65) || defined(__18F86J60)
    || defined(__18F67J60) || defined(__18F66J65) || defined(__18F66J60))
     && !defined(HI_TECH_C)
    // PICDEM.net 2 or any other PIC18F97J60 family device
    #pragma config WDT=OFF, FOSC2=ON, FOSC=HSPLL, ETHLED=ON
#elif defined(_18F97J60) || defined(_18F96J65) || defined(_18F96J60) ||
    defined(_18F87J60) || defined(_18F86J65) || defined(_18F86J60) ||
    defined(_18F67J60) || defined(_18F66J65) || defined(_18F66J60)
    // PICDEM.net 2 board with HI-TECH PICC-18 compiler
    __CONFIG(1, WDTDIS & XINSTDIS);
    __CONFIG(2, HSPLL);
      CONFIG(3, ETHLEDEN);
```

Fig. 2.1 Archivo de configuración de hardware. Fuente: *HardwareProfile.h*

En cuanto a la configuración de la placa, el stack está pensado para su uso conjunto con alguna de las siguientes placas comerciales:

- PIC18_EXPLORER
- HPC_EXPLORER
- DSPICDEM11
- PIC32_STARTER_KIT

Como se puede ver en el código mostrado a continuación:

```
//#define PICDEMNET2
//#define PIC18_EXPLORER
//#define HPC_EXPLORER
//#define PIC24FJ64GA004_PIM
    // Explorer 16, but with the PIC24FJ64GA004 PIM module,
    // which has significantly differnt pin mappings
//#define EXPLORER_16
    // PIC24FJ128GA010, PIC24HJ256GP610, dsPIC33FJ256GP710, PIC32MX360F512L,
    // PIC32MX460F512L, PIC32MX795F512L, etc. PIMs
//#define DSPICDEM11
//#define PIC32_STARTER_KIT
    // PIC32MX360F512L Starter Kit, PIC32MX460F512L USB Starter Board,
    // or PIC32MX795F512L USB Starter Kit II
//#define PIC32_ETH_STARTER_KIT
// PIC32MX795F512L Ethernet Starter Kit board with embedded Ethernet controller
#define YOUR_BOARD
```

Fig. 2.2 Código comentado de placas preconfiguradas. Fuente: *HardwareProfile.h*

Por todo ello es importante destacar la dificultad del uso de un MCU de 8 bits junto con una placa no comercial en un stack, que ya de por sí es complejo.

3. Implementación del hardware

Para llevar a cabo el presente proyecto ha sido necesario el uso de dos MCUs PIC18LF4680 [4] de 8 bits dispuestos cada uno en una placa de prácticas (WaveShare Open18F4520 – Fig.4.1) de las utilizadas en el Grado de Ingeniería de Tecnologías Industriales y en el Máster Universitario de Ingeniería Industrial.

El sensor de temperatura utilizado es el modelo DS18B20 de Maxim Integrated [5]. El funcionamiento de éste y sus características se encuentran en la bibliografía complementaria, ya que su estudio no es objeto de este proyecto.

Asimismo, ha sido necesaria la creación de una pequeña placa de prototipado para incorporar al conjunto una tarjeta de Ethernet con el controlador ENC28J60 [6]. Este controlador ha sido proporcionado por el Departamento de Ingeniería Electrónica de la UPC (sección sud / ETSEIB).

La comunicación entre el microcontrolador PIC18LF4680 y el controlador de ethernet se realiza mediante el bus SPI (Serial Peripheral Interface) de la placa.

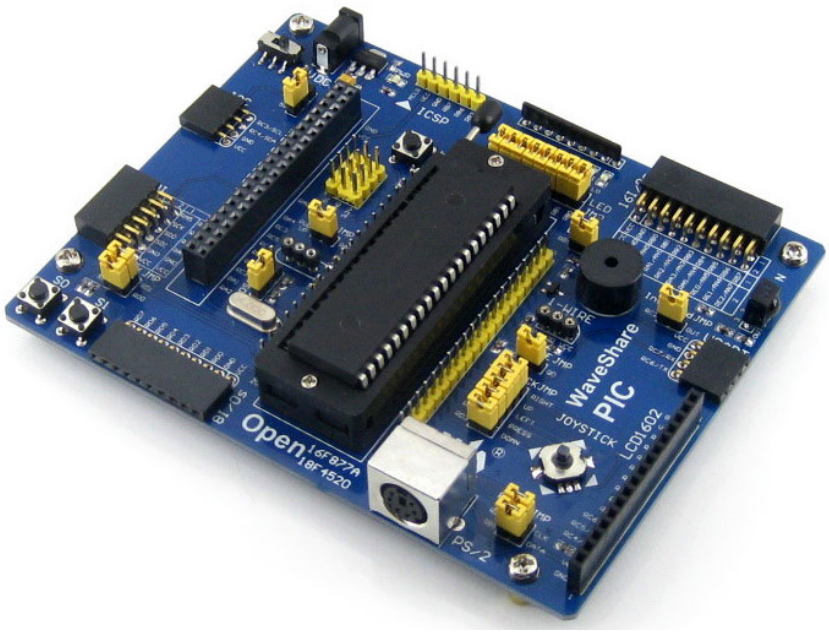


Fig. 3.1 Placa Waveshare Open18F4520. Fuente: [7]

3.1. El microcontrolador PIC18LF4680

El PIC18LF4680 es un microcontrolador (MCU) de 64K bytes de memoria de programa (ROM) y 4K bytes de memoria de datos (RAM) en un encapsulado de 40 pines tipo PDIP. Se trata pues del MCU con mayor memoria para este tipo de encapsulado, dentro de la familia de PIC18.

Se elige el uso de este MCU por su semejanza con el PIC18F4520, utilizado en las prácticas de la intensificación de electrónica del máster de Ingeniería Industrial y por su elevada memoria, que permite alojar el stack de Microchip, ya que como se verá este ocupa mucha memoria.

Si se observa la pirámide OSI (Fig. 2.1), el conjunto PIC18LF4680 / Waveshare Open18F4520 es el encargado de realizar los niveles de comunicación 1 al 5, siendo necesario un controlador de ethernet que se encargue de los dos últimos niveles.

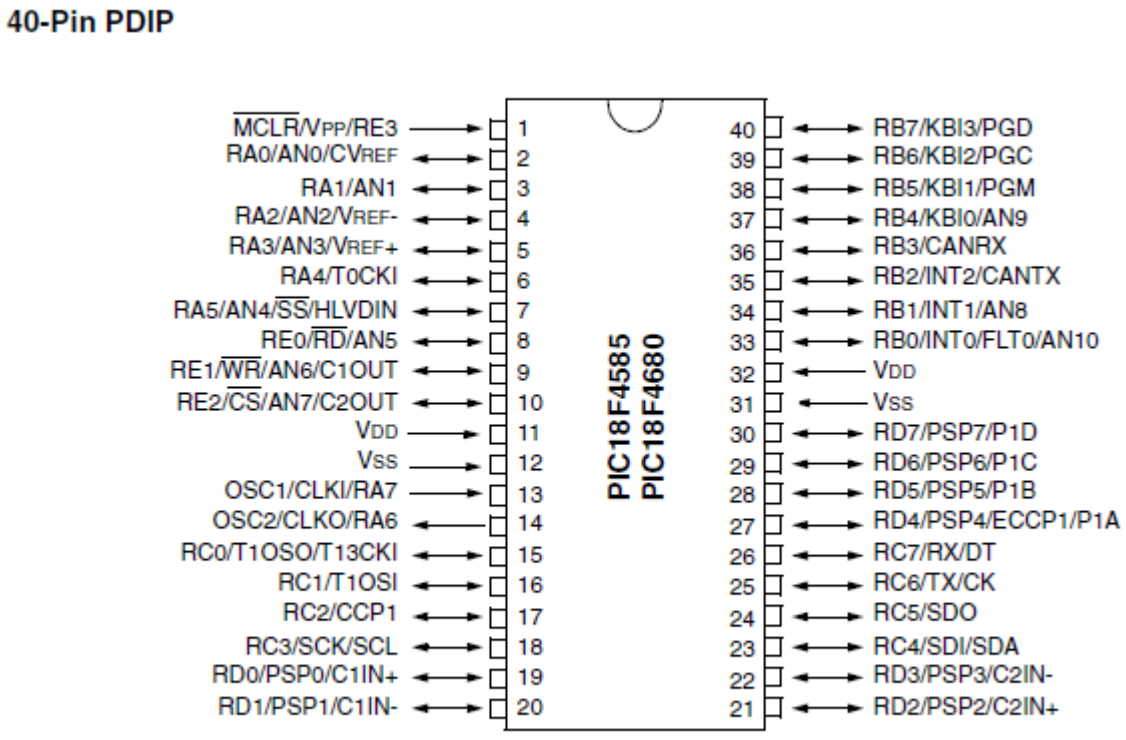


Fig. 3.1 Pines de conexionado del MCU PIC18F4680 . Fuente: [8]

3.2. ENC28J60

El controlador de ethernet utilizado es el ENC28J60 de 28 pines, soldado en la placa comercial denominada ETH Click de Mikroelectronika [10] (véase Fig. 4.2). Se trata de un controlador que gestiona los niveles de enlace de datos y físico (MAC & PHY) de la pirámide OSI. Dispone de un buffer de 8K Bytes de RAM y una interface SPI que utilizamos para conectar con el MCU.
EL controlador se alimenta a 3,3V y utiliza el estándar físico de Ethernet 10Base-T (10Mbps) estándar IEEE 802.3.

Se escoge este controlador por ser uno de los controladores ya integrados en el stack y que solo necesita de una configuración en el archivo de cabecera *HardwareProfile.h.*

Aunque en el presente proyecto no se hace uso de los controladores ENCX24J600, debe notarse que también se hubiesen podido utilizar en cualquiera de sus variantes, debido a su fácil integración en el código del stack. Por sus características, su elección es idónea cuando se requieran aplicaciones en redes de 100Base-T [9] .

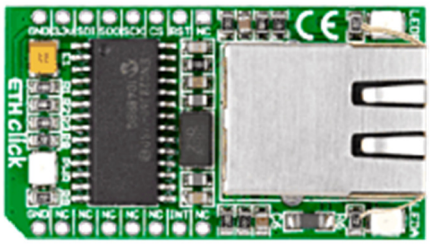


Fig. 3.2 Imagen del controlador de Ethernet ENC28J60. Fuente [11].

Conexión del ENC28J60 al MCU

Para su conexión con el MCU se diseña un circuito sobre una pequeña placa de topos, sobre la que anclaremos la placa Ethernet Click.

Para realizar dicho diseño se deben conocer qué pines hacen uso el stack. Una búsqueda en el archivo de cabecera del stack *HardwareProfile.h* muestra los registros a los que accede éste y por tanto, los pines que debemos conectar por SPI. El funcionamiento de este archivo de configuración y del stack se muestra en detalle en el siguiente capítulo.

```
// ENC28J60 I/O pins
#define ENC_RST_TRIS        (TRISBbits.TRISB5)
#define ENC_RST_IO          (LATBbits.LATB5)
#define ENC_CS_TRIS         (TRISBbits.TRISB3)
#define ENC_CS_IO           (LATBbits.LATB3)
#define ENC_SCK_TRIS        (TRISCbits.TRISC3)
#define ENC_SDI_TRIS        (TRISCbits.TRISC4)
#define ENC_SDO_TRIS        (TRISCbits.TRISC5)
#define ENC_SPI_IF          (PIR1bits.SSPIF)
#define ENC_SSPBUF          (SSP1BUF)
#define ENC_SPISTAT         (SSP1STAT)
#define ENC_SPISTATbits     (SSP1STATbits)
#define ENC_SPICON1         (SSP1CON1)
#define ENC_SPICON1bits     (SSP1CON1bits)
#define ENC_SPICON2         (SSP1CON2)
```

Fig. 3.2 Configuración de los registros de la placa utilizada. *HardwareProfile.h*

Analizado el archivo de configuración y conociendo la disposición de los pines del puerto SPI de la placa, las pistas a soldar quedan de la siguiente manera:

ENC28J60	SPI PLACA	PISTA
GND	GND	1
+3,3V	VCC	2
SCK	SCK	3
CS	RA3	4
SDI	SDO	5
SDO	SDI	6

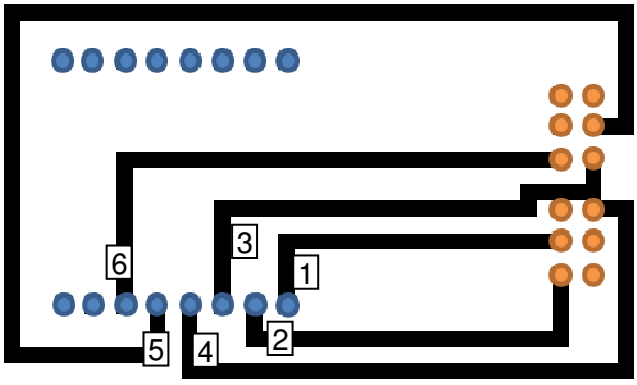


Fig. 3.3 Esquema de conexión de la controladora Ethernet al puerto SPI de la placa.
Fuente: Propia

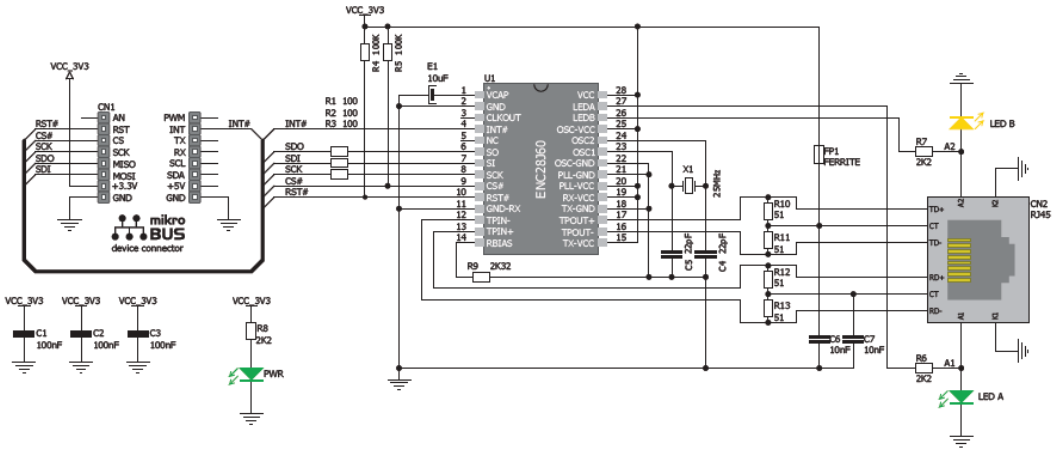


Fig. 3.4 Esquema placa Ethernet Click con controlador Ethernet ENC28J60.
Fuente: [10]

3.3. Retos a superar en la implementación del hardware

El desarrollo de un proyecto que requiera de la utilización y/o desarrollo de hardware específico supone un reto suplementario, debido a que los errores en el hardware son muchas veces más difíciles de detectar que los errores en el software. Es más fácil y supone un menor coste disponer de herramientas para detectar errores de compilación, analizar registros y poner puntos de ruptura (*breakpoints)* en partes del código para analizar su correcto funcionamiento.

Es por ello que a continuación se enumeran y explican las dificultades que han aparecido a lo largo del proyecto, provocando en ocasiones tener que repetir la placa de conexionado.

PIC18LF4680 vs PIC18F4680

El microcontrolador utilizado durante una buena parte del desarrollo no fue el modelo PIC18LF4680, sino el PIC18F4680 proporcionado a tal efecto al iniciar el proyecto.

No fue hasta que se detectaron varios errores en los registros, que se planteó la solución de reducir la velocidad de reloj del PIC. Esto se debe a que el controlador de ethernet ENC28J60 funciona a 3,3V, por lo que el PIC también estaba alimentado a 3,3V. Este hecho hacía necesario bajar la frecuencia, tal y como indica la ficha técnica de especificaciones (Fig. 3.6). Para ello se modificó el archivo *HardwareProfile.h* y se cambió la configuración de HSPLL a HS (desactivación de la PLL interna del microntrolador), cambiando a su vez el *HTSystemCLock()* a 4MHz.

```
    #else
        #define GetSystemClock()     (4000000ul)      // Hz
        #define GetInstructionClock()   (GetSystemClock()/4)
        #define GetPeripheralClock()    GetInstructionClock()
    #endif
#pragma config OSC=HS, WDT=OFF, MCLRE=ON, PBADEN=OFF, LVP=OFF
```

Fig. 3.5 Configuración en código del nuevo reloj. Fuente: *HardwareProfile.h*.

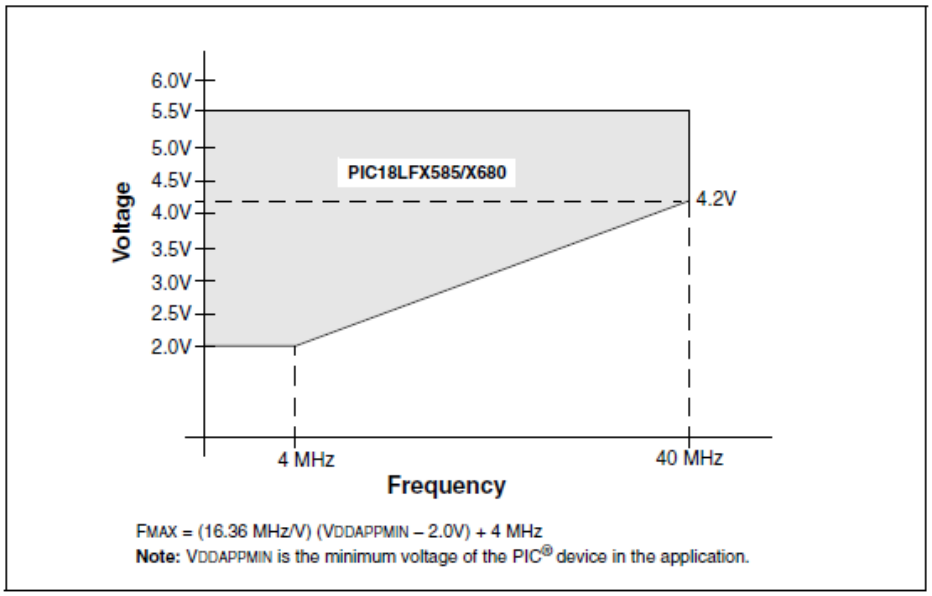


Fig. 3.6 Curva de frecuencia de trabajo del PIC18LF4680. Fuente: [12]

Aunque se había disminuido la velocidad del reloj en código, los problemas continuaron hasta que, repasando las especificaciones, se dio con la gráfica (Fig. 3.7), donde se muestra que la versión del PIC utilizada no es compatible con la disminución de tensión aplicada. Este hecho hizo que se debiese cambiar otra versión del PIC compatible con baja tensión (modelo LF).

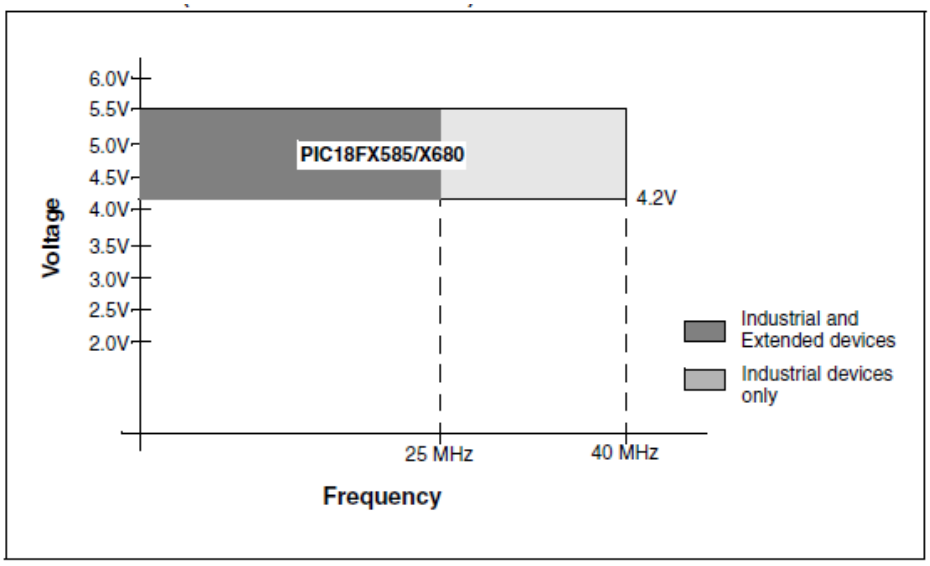


Fig. 3.7 Curva de frecuencia de trabajo del PIC18F4680. Fuente:[12]

Conversor A/D

El siguiente problema estuvo presente durante más de tres meses, llegando casi a imposibilitar la obtención de los resultados finales. Se trata de un error intrínseco al planteamiento que hace Microchip en su stack para la obtención de números aleatorios.

Como se explicará con mayor detalle en el apartado 4.4 Aplicación principal - *MainDemo.c*, en el bucle principal del código se plantea el uso de un led intermitente que ayude a visualizar la sincronía del stack. La frecuencia de parpadeo de este led es constante, ya que viene determinado por una función que depende directamente del reloj del MCU (*GetTick()*), por lo que cualquier desviación perceptible es un signo inequívoco de que el stack no está funcionando correctamente.

Durante las pruebas de uso y desarrollo del código podía percibirse que la placa y el MCU dejaban de funcionar, en lo que parecía un error aleatorio. Dejaba de poder acceder a las distintas funciones del stack, siendo el caso más claro el que se producía cuando el MCU dejaba de responder al solicitar la página web que tenía guardada en memoria. Esto sucedía en lo que, repito, parecía un suceso aleatorio.

Después de inspeccionar todas las partes del código, simplificarlo a su mínima expresión y volviendo a versiones guardadas en las que se sabía seguro que el código funcionaba correctamente, se pudo descartar que no fuese un problema de software.

Para encontrar el causante del problema se intentaron reproducir las condiciones en las que aparecía el fallo. A medida que aparecía el problema se notó que su repetitividad estaba afectada por el conexionado de la placa de topos con el SPI.

De esta manera se pasó a comprobar la lógica del conexionado y la continuidad de este con un tester, repitiendo incluso una de las placas para descartar cualquier error. Finalmente, no siendo capaz de encontrar el error, se analizaron las señales durante el funcionamiento con el uso de un osciloscopio. No se encontraron interferencias o señales fuera de especificaciones.

Durante las pruebas sí que se encontró una mejora en el comportamiento del parpadeo del led al tocar el pin RA1 del SPI. El pin RA1 del SPI no está conectado a la tarjeta de Ethernet, como se puede comprobar en el esquema de la Fig. 3.1. Este conector corresponde al registro RA1 en el código del stack ADCON0.

A pesar de que se trata de una entrada al aire, si se hace una búsqueda entre todas las líneas de código donde se utiliza esta variable en el código, nos encontramos, entre otros, con la siguiente función: *GenerateRandomDWORD(void)*

En esta función, perteneciente al código *Helpers.c*, se utiliza el conversor A/D del microcontrolador para captar aleatoriedad y utilizarla en otra funciones (*srand() / rand() / nextport()*) tal y como se observa en la figura 4.8.

```
Function:
  DWORD GenerateRandomDWORD(void)

Summary:
  Generates a random DWORD.

Description:
  This function generates a random 32-bit integer.  It collects
  randomness by comparing the A/D converter's internal R/C oscillator
  clock with our main system clock.  By passing collected entropy to the
  C rand()/srand() functions, the output is normalized to meet statistical
  randomness tests.
```

Fig. 3.8 Descripción de la función de generación de números aleatorios implementada en *Helpers.c*. Fuente: *Helpers.c*

Para evitar los problemas que causaba la adquisición de esta señal, necesaria para la obtención de números aleatorios y para evitar interferencias con otras partes del stack, como el módulo *Tick.c* que es esencial para el funcionamiento correcto del stack, se optó por la solución que ha demostrado mejores resultados. Aumentar la antena del canal 1 (RA1) del conversor A/D (Fig. 3.1)

```
Side Effects:
  This function uses the A/D converter (and so you must disable
  interrupts if you use the A/D converted in your ISR).  The C rand()
  function will be reseeded, and Timer0 (PIC18) and Timer1 (PIC24,
  dsPIC, and PIC32) will be used.  TMR#H:TMR#L will have a new value.
  Note that this is the same timer used by the Tick module.
```

Fig. 3.3. Descripción de la interacción de la función con el modulo Tick. Fuente: *Helpers.c*

4. Implementación del Software

4.1. Paquetes de Software utilizados

Para el diseño, programación, gestión y control de las aplicaciones diseñadas se hace uso de los siguientes paquetes de software:

WireShark v2.0

Se trata de un *sniffer* o programa capturador de paquetes. Se utiliza para analizar el uso y funcionamiento de distintos protocolos de comunicación, al poder capturar e inspeccionar distintos paquetes que circulan por la red a la que esté conectado el PC.

Python v2.7

Lenguaje de programación de alto nivel y fácil sintaxis. Se elige por ser actualmente utilizado en las prácticas del Grado y del Máster Universitario y su facilidad de uso para la creación de sockets de comunicación.

MPLAB v8.63 IDE

Entorno de programación para el desarrollo de aplicaciones de microcontroladores de Microchip. Las aplicaciones de este proyecto se desarrollan en C y se compilan con el compilador MPLAB C18, que sigue el estándar ANSI 89, en su versión gratuita v3.36.

Microchip Applications Library (MAL)

Librerías gratuitas proporcionadas por Microchip en las que se encuentran proyectos, demostraciones, documentación, utilidades y código para la realización de aplicaciones. En estas librerías encontramos el stack de protocolos de comunicación que utilizaremos para el desarrollo de las máquinas, así como una serie de APIs que facilitan su uso y configuración.

Hasta febrero de 2016 existen 23 versiones de la librería, lo que demuestra la complejidad del desarrollo que se lleva a cabo.

Debido al uso del MPLAB v8.63, únicamente podremos utilizar versiones anteriores a la 11/2013. Para el desarrollo de las aplicaciones se ha trabajado con la versión del stack y la APIs provistas en las MALs v18-11-2009.

4.2. Introducción al Stack de Microchip y sus partes

Introducción al Stack

El stack de Microchip se trata de un conjunto de rutinas y subrutinas escritas en lenguaje C que constituyen el marco base sobre el cual se pueden diseñar y programar aplicaciones que utilicen alguno de los principales protocolos de comunicación o algunas de sus características.

Entre los protocolos disponibles se encuentran los siguientes:

> **Protocolos a nivel aplicación:** Web Server (con todas las características que ofrece HTTP2), E-mail Clients, Telnet, SNMP, TFTP.
>
> **Protocolos a nivel transporte**: TCP, UDP, SSL
>
> **Protocolos a nivel de red**: IP
>
> **Protocolos a nivel de enlace de datos**: ARP, MAC

Además, permite el uso de diversos módulos para su implementación con alguno de estos protocolos de aplicación, como clientes DHCP, IP Gleaning, servicios de ICMP (Ping) o clientes DNS.

El funcionamiento del stack es modular. Podemos configurar qué protocolos utilizará nuestra aplicación.

Los protocolos están interconectados. Funciones de alto nivel de la pirámide OSI (nivel de aplicación) harán llamadas a funciones de niveles inferiores hasta recorrer todos los niveles.

Siendo únicamente el stack un marco base sobre el que se debe desarrollar/implementar una aplicación, todas las implementaciones de los protocolos que se hacen deben funcionar como máquinas de estado colaborativas, que van cediendo ciclos de cálculo a otras partes del stack o de la aplicación principal [3].

Esto queda explicado en mayor profundidad al analizar el archivo principal sobre el que debemos construir nuestra aplicación. *MainDemo.c.*

Organización del Stack

Para facilitar la compresión de la estructura organizativa del stack, parece conveniente dividirlo en dos, según el uso que se hace del código.

En una primera parte encontramos los archivos de *bajo nivel del stack*, que recogen cada una de las funciones y los estados de las máquinas con los que se ha definido el funcionamiento de cada protocolo.

Esta primera parte la constituyen 47 archivos. Entre ellos están todos los protocolos anteriormente descritos en la introducción, así como otros archivos de código que sirven para la gestión de aspectos del stack, como las controladoras de Ethernet, las definiciones para la inclusión de una página web embebida (MPFS) y diversas ayudas y definiciones (helpers).

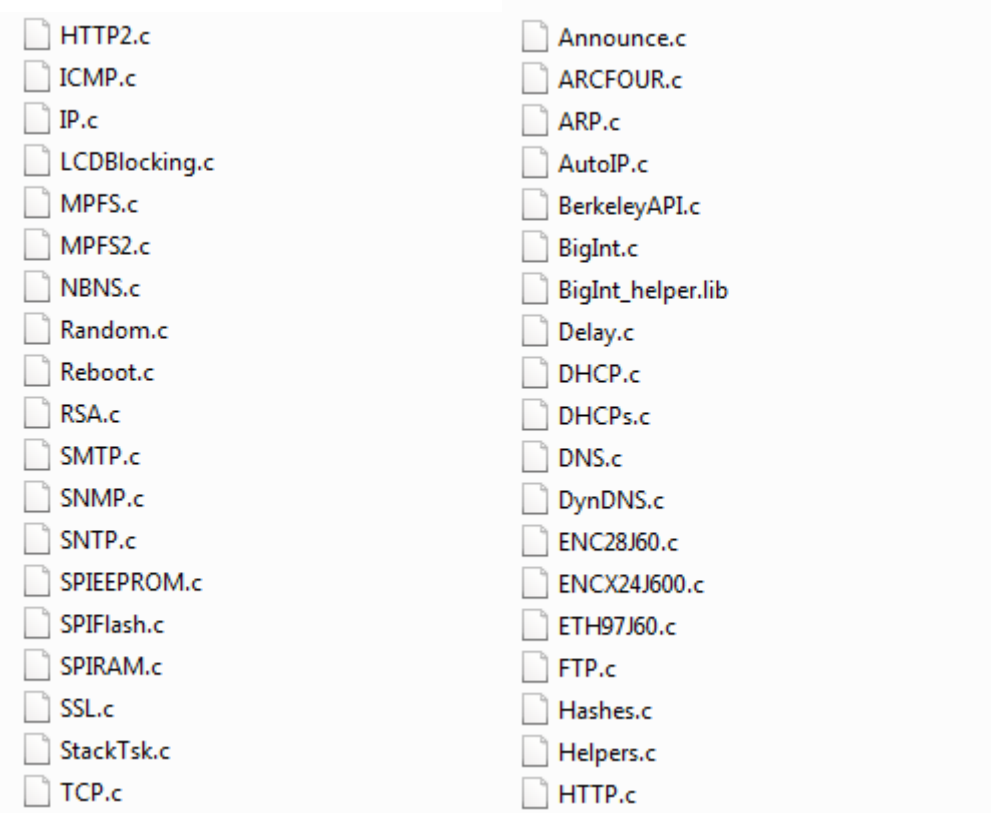


Fig. 4.1 Archivos de código base del stack. Fuente: Propia

Entre todos ellos suman más de **46.000 líneas de código**, sin contar los archivos de cabecera.

Este proyecto no persigue la modificación de los archivos que conforman la base del stack sino su uso y compresión dentro de una aplicación con fines didácticos para los estudiantes de ingeniería.

La segunda parte del stack lo conforman los archivos *de alto nivel del stack*. Son aquellos archivos encargados de:

- La configuración del hardware – (*HardwareProfile.h):* Se trata del archivo de cabecera donde se han configurado las características de nuestra placa y microcontrolador. Están descritas en el apartado dedicado al hardware.

- La configuración del propio Stack – (*TCPIPConfig.h):* Se trata del archivo de cabecera donde se especifica qué partes/protocolos del stack van a ser utilizados. Su uso y configuración es explicado en el siguiente apartado. Microchip proporciona una API para facilitar configuración.

- Desarrollo de la aplicación principal - *(MainDemo.c):* Este es el archivo principal donde se programa el uso y configuración, explicado en el siguiente apartado.

- Desarrollo de subrutinas - (*Ping.c, GenericTCPClient.c, CustomHTTP.c, BroadcastUDP.c ...):* Se trata de las subrutinas de comunicación o funciones HTTP que dan soporte a la aplicación principal. En el presente proyecto se diseñan una o más subrutinas para cada una de las máquinas.

- Inclusión de una página Web - *(MPFSImg2.c):* Este archivo es la manera que utiliza el stack para guardar páginas web. Cualquier conjunto de páginas web que queramos que sean servidas por un aplicativo de servidor web, deberán convertirse mediante una aplicación proporcionada por Microchip (MPFS2.exe).

4.3. Configurar el STACK- *TCPIPConfig.h*

La configuración se realiza con el archivo de cabecera *TCTIPConfig.h*. En este archivo encontramos:

- Módulos a nivel de aplicación. El stack proporciona 27 módulos distintos, que van desde el uso de DNS a la creación de un servidor TELNET, así como diversas opciones de configuración anteriormente descritas. Se comentan y descomentan (//) las líneas de código para activar las diferentes opciones de configuración:

```
// ============================================================================
//   Application Options
// ============================================================================

/* Application Level Module Selection
 *   Uncomment or comment the following lines to enable or
 *   disabled the following high-level application modules.
 */
#define STACK_USE_UART                  // Application demo using UART for
                                        //  IP address display and stack
                                            //configuration
//#define STACK_USE_UART2TCP_BRIDGE     // UART to TCP Bridge application example
//#define STACK_USE_IP_GLEANING
//#define STACK_USE_ICMP_SERVER         // Ping query and response capability
//#define STACK_USE_ICMP_CLIENT         // Ping transmission capability
//#define STACK_USE_HTTP_SERVER         // Old HTTP server
```

Fig. 4.2 Módulos del stack desactivados Fuente: *TCPIPConfig.h*

- La configuración MAC / IPv4 de nuestra máquina (ENC28J60 + MCU). Donde debemos introducir la dirección MAC (en caso de que no queramos utilizar la predeterminada por el fabricante), dirección IP, puerta de enlace, mascara de subred y direcciones DNS.

```
#define MY_DEFAULT_HOST_NAME        "MCHPBOARD"

#define MY_DEFAULT_MAC_BYTE1        (0x00)
#define MY_DEFAULT_MAC_BYTE2        (0x04)
#define MY_DEFAULT_MAC_BYTE3        (0xA3)
#define MY_DEFAULT_MAC_BYTE4        (0x00)
#define MY_DEFAULT_MAC_BYTE5        (0x00)
#define MY_DEFAULT_MAC_BYTE6        (0x00)

#define MY_DEFAULT_IP_ADDR_BYTE1    (169ul)
#define MY_DEFAULT_IP_ADDR_BYTE2    (254ul)
#define MY_DEFAULT_IP_ADDR_BYTE3    (1ul)
#define MY_DEFAULT_IP_ADDR_BYTE4    (1ul)
```

- Configuración de la encriptación WiFi. El stack es capaz de utilizar encriptación WEP / WAP y WAP2. La comunicación por WiFi no es objeto de estudio de este proyecto.

- Configuraciones específicas de alguno de los módulos de aplicación (HTTP2, Telnet, SSL, SNMP).

- Configuración de los módulos a nivel de transporte (TCP o UDP) y la configuración de los sockets de comunicación. Este último apartado es interesante, ya que el espacio de memoria RAM que van a utilizar los sockets no se gestiona dinámicamente y debe de ser configurado previamente a su uso.

```
// Allocate how much total RAM (in bytes) you want to allocate
// for use by your TCP TCBs, RX FIFOs, and TX FIFOs.
#define TCP_ETH_RAM_SIZE                    (4509ul)
#define TCP_PIC_RAM_SIZE                    (0ul)
#define TCP_SPI_RAM_SIZE                    (0ul)
#define TCP_SPI_RAM_BASE_ADDRESS            (0x00)


#define TCP_PURPOSE_GENERIC_TCP_CLIENT 0

{TCP_PURPOSE_LED_SERVER, TCP_PIC_RAM, 200, 20}
```

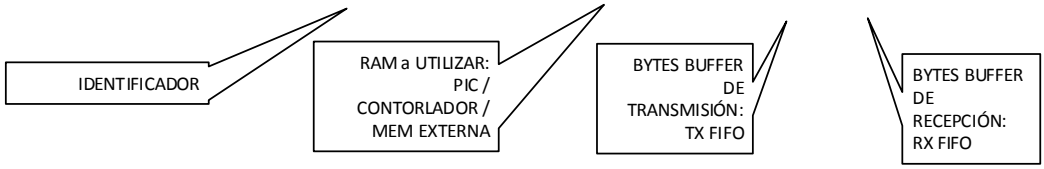


Fig. 4.3 Configuración de un socket de conexión TCP. Fuente: Propia

4.4. Aplicación principal - *MainDemo.c*

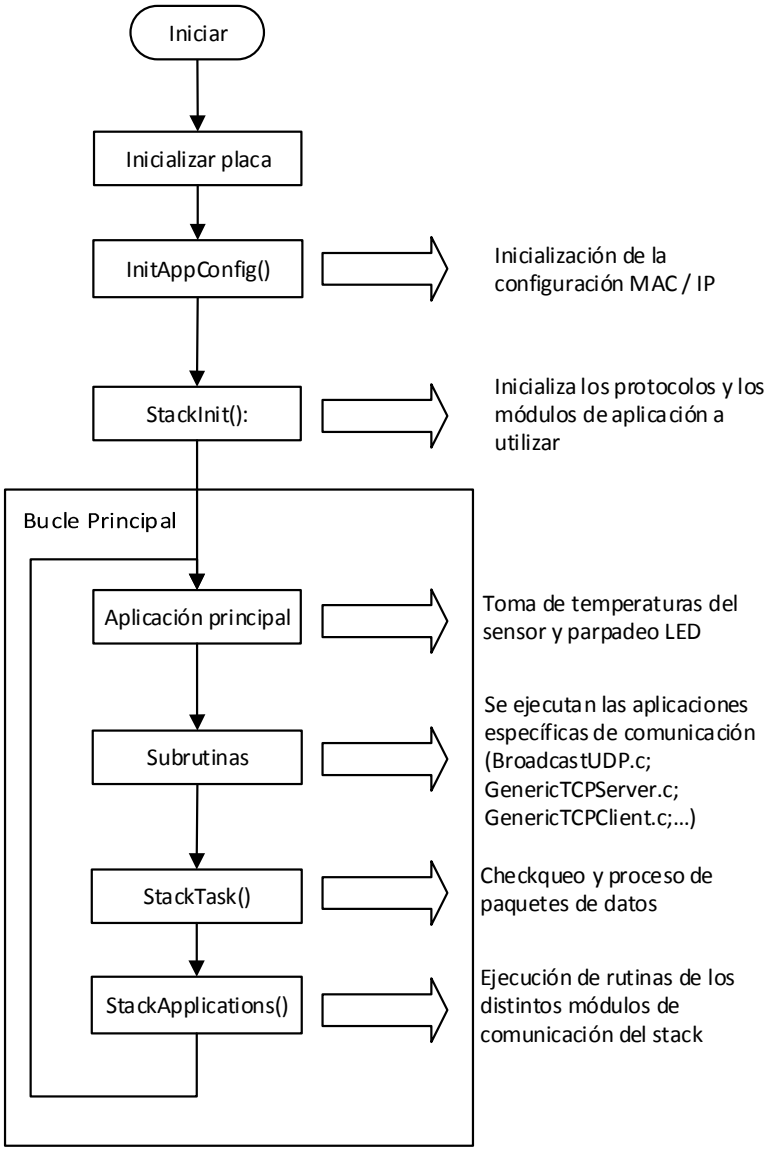


Fig. 4.4 Máquina de estados de *MainDemo.c* Fuente: *Propia*

Las distintas aplicaciones construidas en este proyecto siguen la misma estructura mostrada en la Fig. 4.4. La inicialización del archivo *MainDemo*.c lo constituyen más de 500 líneas de código, muestra de las múltiples capas de hardware y software que deben inicializarse.

A partir de aquí se entra en un bucle infinito, donde debe ejecutarse la aplicación principal, las subrutinas que utilice nuestro stack (BroadcastUDP.c; GenericTCPServer.c; GenericTCPClient.c) y las tareas propias de bajo nivel del stack.

Cualquier aplicación que utilice el stack deberá tener en cuenta que no podrá hacer uso del todos los ciclos de cálculo, por lo que deberá de desarrollarse como una máquina de estados que deje ciclos de cálculo para el resto del stack. Asimismo, si nuestra aplicación principal requiriese de elevados tiempos de proceso, también debería implementarse como una máquina de estado colaborativa que dejase tiempo de cálculo para el resto de tareas.

La aplicación principal: Captura de temperaturas con el sensor DS18B20 y parpadeo LED.

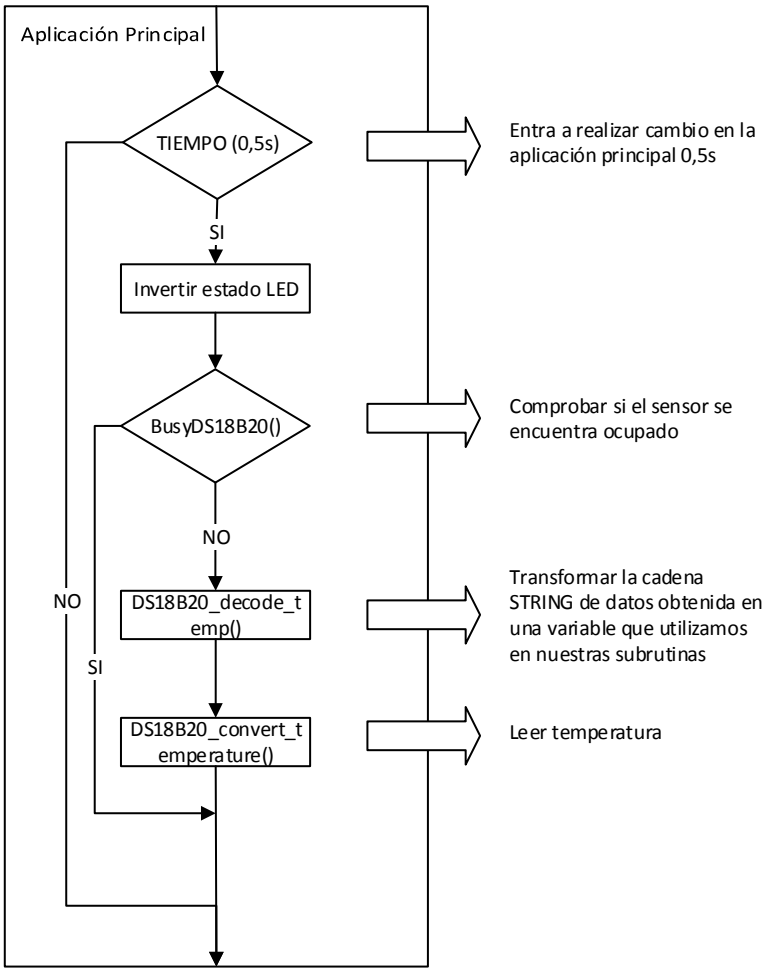


Fig. 4.5 Máquina de estados de la aplicación principal. Fuente: *Propia*

La aplicación principal será la misma en todas las aplicaciones presentadas en el proyecto. Consta de dos funcionalidades, siendo la primera de ellas una captura de temperaturas utilizando las funciones descritas en el manual DS18B20 Lib User Manual [13] adjuntado en la bibliografía adicional.

```
if(TickGet() - t >= TICK_SECOND/2ul)
{
    t = TickGet();
    LED0_IO ^= 1;
    numero=numero+1;
    //while(BusyDS18B20());
    if(!BusyDS18B20())
    {
        // Read Temp
        if((DS18B20_read_scratchpad(scratchpad)==1))
        {

            DS18B20_decode_temp(scratchpad, TEMP1);
            Actu1=0;
            Final1=1;
        }
        // Convert Temp
        DS18B20_convert_temperature();
    }

    if(numero > 30)
{
    Actu1=1;
    Actu2=1;
    numero=0;
}
```

Fig. 4.6 Código de la aplicación MainDemo.c encargado de la toma de temperaturas y el parpadeo del LED Fuente: MainDemo.c

5. MAQ1-Servidor Web

5.1. Introducción

En esta primera aplicación se va a analizar el funcionamiento del protocolo HTTP (Nivel Aplicación) y su implementación en un MCU de la familia PIC18 mediante el stack de Microchip.

Para ello se implementará un servidor Web en un microcontrolador (MCU1) conectado a un PC creando una red ad-hoc, tal y como se muestra en la Figura 6.1. El MCU1 gestionará un servidor web, lo que permitirá mostrar por cualquier navegador la temperatura del sensor. La temperatura se mostrará utilizando una variable dinámica dentro del código HTML de la web alojada en el servidor. La actualización de esta variable dinámica es regida por un temporizador en el MCU1.

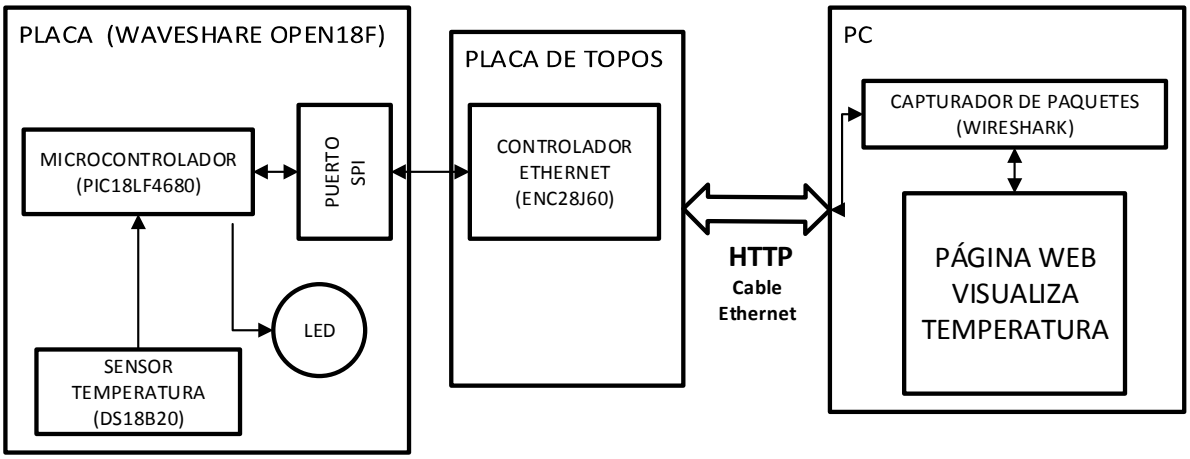


Fig. 5.1 Esquema de conexionado. Fuente: Propia.

El stack permite el uso de funciones avanzadas de HTTP (envíos de formularios GET/POST, autentificación de credenciales, subidas de archivos desde HTTP, cookies,..). Debido a que el factor limitador del servidor es la memoria del MCU, solo se han utilizado métodos GET y variables dinámicas.

5.2. Configuración del STACK

El primer paso es la configuración del stack. Para economizar la poca memoria RAM del MCU se desactivan todos módulos menos el de HTTP. Al ser un stack modular, la activación de éste activa el resto de módulos de bajo nivel necesarios para establecer la conexión (TCP; ARP; IP; MPFS; ...).

Configuramos la dirección IP del siguiente modo

IP Address: 169.254.1.1
Gateway: 169.254.1.148
DNS1: 169.254.1.148

Al tratarse de un servidor de una red ad-hoc solo es necesaria la dirección IP de éste, sin requerir puerta de enlace o DNS.

Finalmente configuramos el socket de conexión. Se utiliza la RAM del controlador Ethernet (TCP_ETH_RAM) con un buffer de transmisión (TX) de 500 bytes y uno de recepción (RX) de 500 bytes. Son dos buffers pequeños para los estándares que recomienda Microchip. En este caso y debido a los pocos bytes de la web a transmitir no aumentará el retraso (lag).

```
#define TCP_PURPOSE_HTTP_SERVER 8

{TCP_PURPOSE_HTTP_SERVER, TCP_ETH_RAM, 500, 500},
```

5.3. Implementación del código HTML

A continuación, se diseña el código HTML de la web, con la variable dinámica *~temperatura1~* y un pequeño programa en JavaScript que el navegador interpretará como un botón de carga. *window.location.reload(true)*

```
<head></head>
▼<body>
  <h1>T: ~temperatura1~    ºC</h1>
  ▼<p>
    <input onclick="window.location.reload(true)" type="button" value="Actualitza">
  </p>
</body>
</html>
```

Fig. 5.2 Código HTML utilizado en la página web. Fuente: Temperatura.htm

Una vez acabado el código es necesario introducir la web dentro del microcontrolador e implementar la variable dinámica *~temperatura1~* en el código del stack (Fig. 5.2).

Para ello, la aplicación proporcionada por Microchip dentro del stack (MPFS2.exe) convierte el archivo con extensión html en una imagen MPFS. Además, genera un archivo de encabezado con las variables dinámicas utilizadas. En este caso *HTTPPrint_temperatura1();*

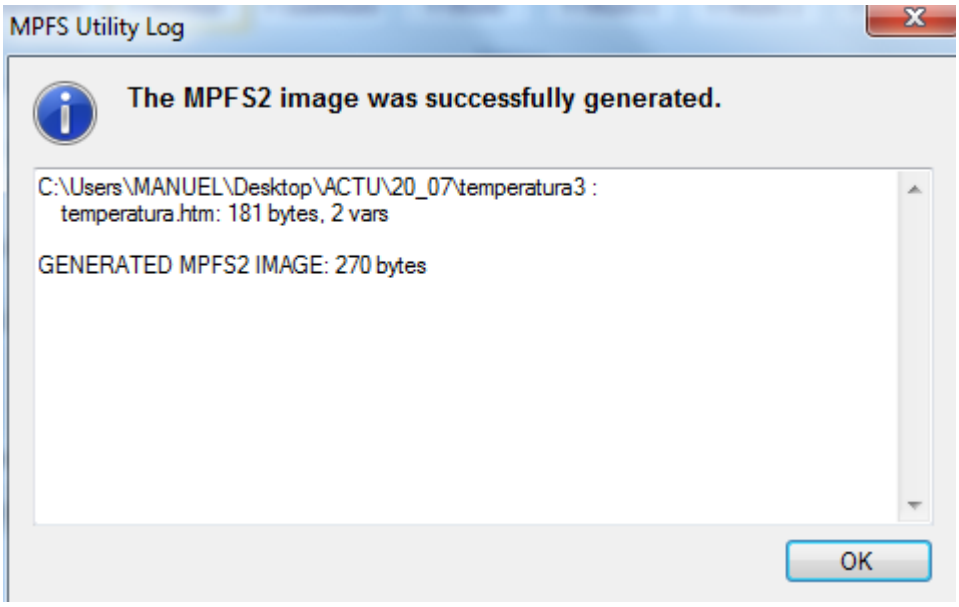


Fig. 5.3 Página HTML *temperatua.htm* convertida a formato MPFS2. Fuente: Captura de MPFS2.exe

Uso de variables dinámicas

Todas las funciones de las que haga uso la web deben de implementarse dentro del archivo *CustomHTTPApp.c*

La implementación de una variable dinámica siempre sigue el mismo formato:

```
void HTTPPrint_variable_dinamica(void)
{
TCPPutString(sktHTTP,variable_put);
}
```

HTTPPrint_variable_dinamica → Es el nombre de la variable que ejecutará el navegador.

TCPPutString→ La función de envío (PUT) que se utiliza para transmitir la variable dinámica. Es una de las funciones de nivel de transmisión TCP que contiene el protocolo.

Variuable_put→ Es la variable de tipo STRING, cuyo valor será enviado por el MCU al navegador.

sktHTTP→ Es el socket *TCP_PURPOSE_HTTP_SERVER,* que se abre en la función *HTTPInit* (), mostrada a continuación, por el que se enviará la variable.

```
void HTTPInit(void)
{
    PTR_BASE oldPtr;

    // Make sure the file handles are invalidated
    curHTTP.file = MPFS_INVALID_HANDLE;
    curHTTP.offsets = MPFS_INVALID_HANDLE;

    for(curHTTPID = 0; curHTTPID < MAX_HTTP_CONNECTIONS; curHTTPID++)
    {
        smHTTP = SM_HTTP_IDLE;
        sktHTTP = TCPOpen(0, TCP_OPEN_SERVER, HTTP_PORT, TCP_PURPOSE_HTTP_SERVER);
        #if defined(STACK_USE_SSL_SERVER)
        TCPAddSSLListener(sktHTTP, HTTPS_PORT);
        #endif
```

Fig. 5.4 Parte del código de inicialización del servidor web. Fuente: *HTTP.c*

La función *TCPOpen*, que sirve para abrir el socket de conexión, forma parte del protocolo TCP (Nivel 4 de la pirámide OSI) y es mostrada en la MAQ3 de este proyecto.

La implementación de la variable queda de la siguiente forma:

```
void HTTPPrint_temperatura1(void)
{
    TCPPutString(sktHTTP,TEMP1);
}
```

Fig. 5.5 Variable dinámica implementada en *CustomHTTP.c* Fuente: *CustomHTTP.c*

En este caso, la variable TEMP1 es donde está guardada la temperatura en formato STRING después de ejecutar la subrutina *DS18B20_decode_temp(scratchpad, TEMP1)* en el bucle principal de la aplicación.

5.4. Conexión y Funcionamiento

Se utiliza el proyecto MAQ1-Web Server.mcw volcado en el MCU1.

Una vez finalizada la compilación, se procede a conectar el servidor web a un PC y se analiza el flujo de datos con WireShark. En esta primera máquina, la gestión de la conexión la realiza la máquina de estados HTTP del stack, por lo que no es objeto de esta aplicación analizar los módulos ARP y TCP.

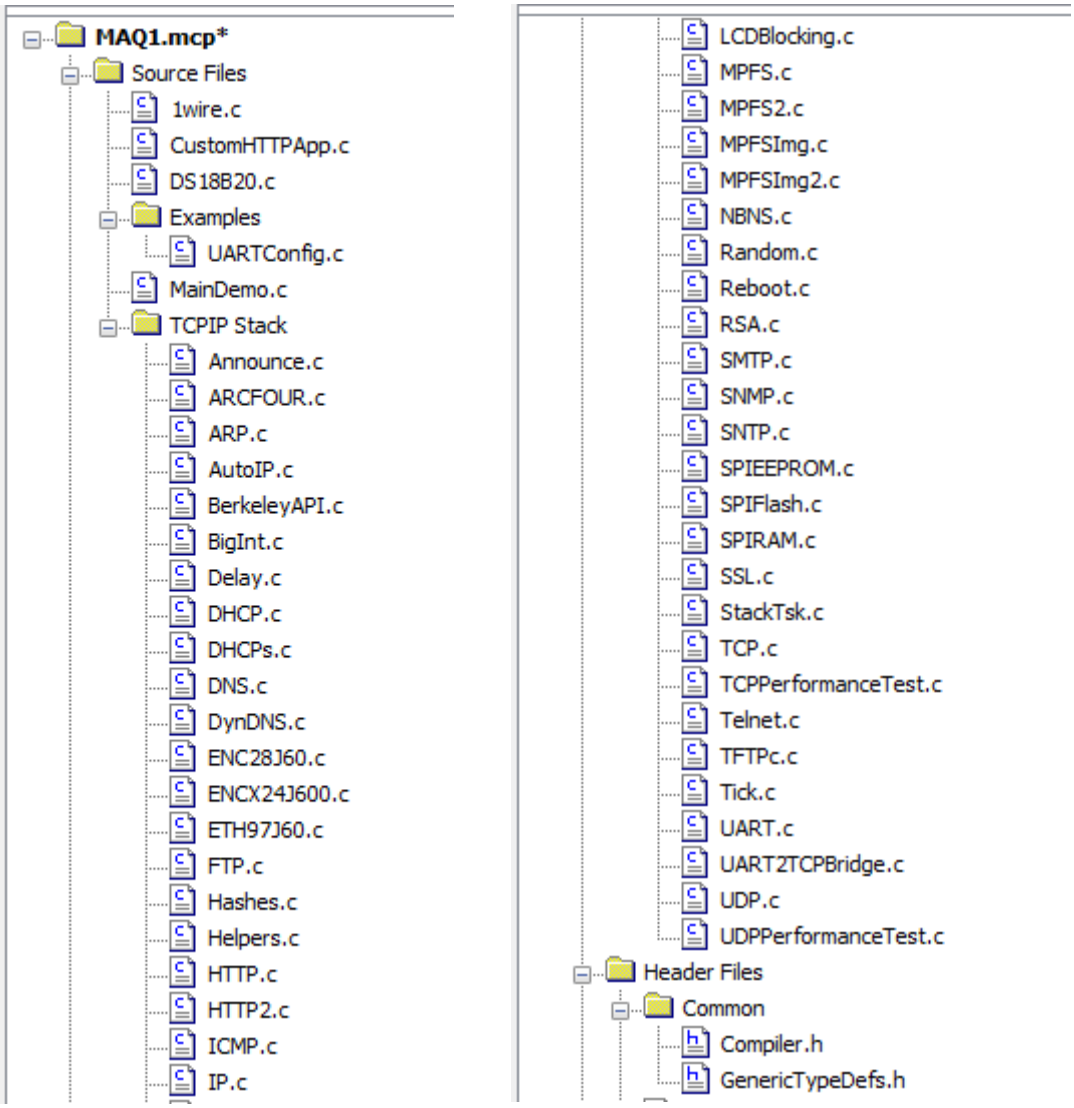


Fig. 5.6 Esquema de archivos del proyecto. Fuente: MAQ1-WebServer.mcw

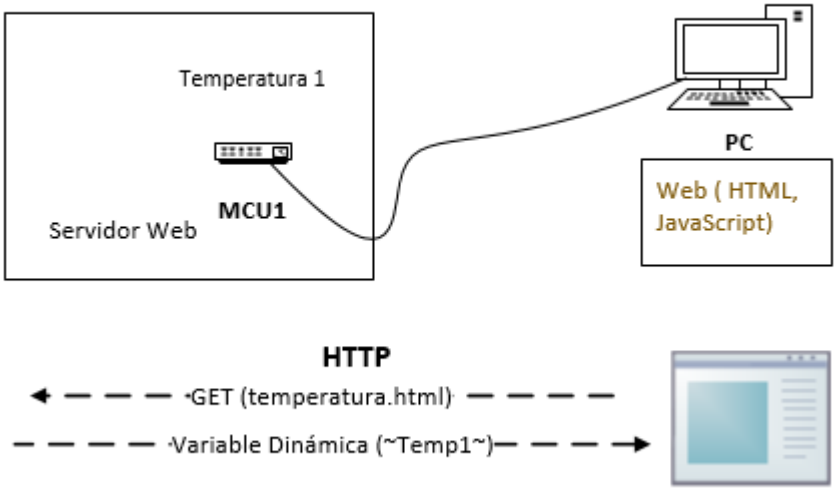


Fig. 5.7 Esquema de funcionamiento. Fuente: Propia

Se accede a la web mediante la dirección URL http://169.254.1.1/temperatura.htm

Esto queda reflejado en la siguiente captura de pantalla de WireShark, donde se han señalado en negro los paquetes HTTP correspondiente de nuestra aplicación.

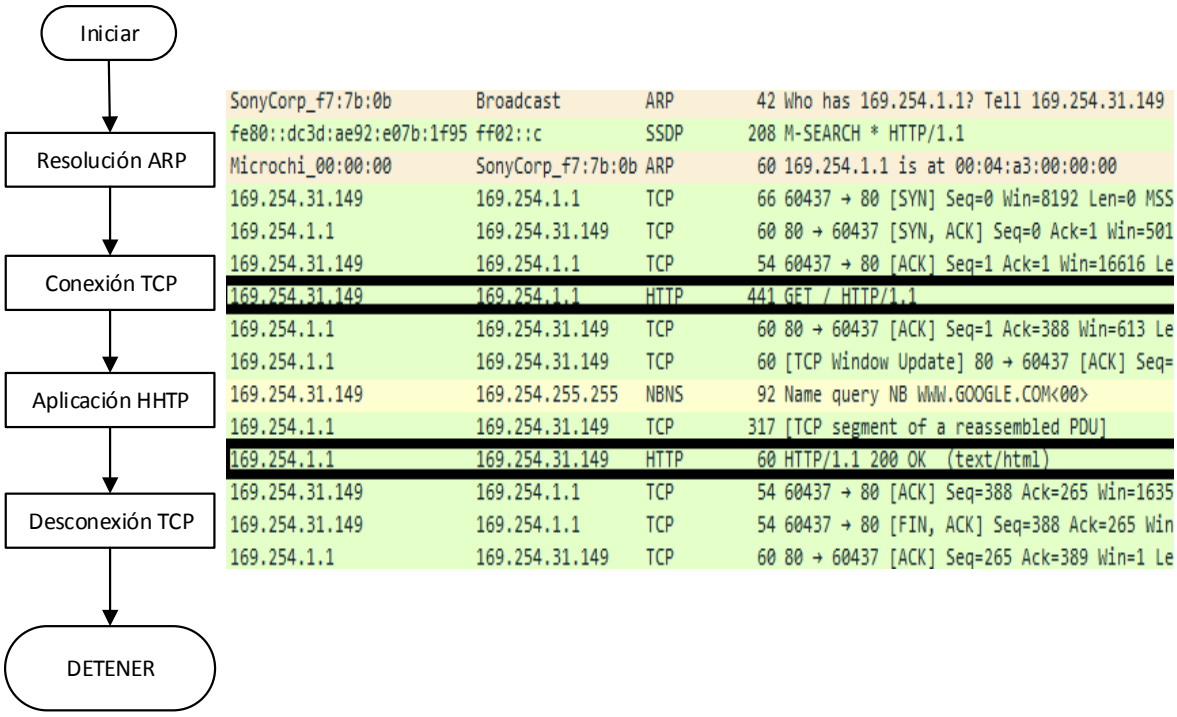


Fig. 5.8 Paquetes captados por WireShark Fuente: Propia.

Si ahora se comparan estos dos paquetes HTTP recogidos por WireShark con los estados de la máquina de estados con los que se ha definido el protocolo HTTP, se observa lo siguiente que de todos los estados de HTTP solo se procesan HTTP_SERVE BODY y HTTP_SEND_FROM_CALLBACK

```
SM_HTTP_IDLE = 0u,
SM_HTTP_PARSE_REQUEST,
SM_HTTP_PARSE_HEADERS,
SM_HTTP_AUTHENTICATE,
SM_HTTP_PROCESS_GET,
SM_HTTP_PROCESS_POST,
SM_HTTP_PROCESS_REQUEST,
SM_HTTP_SERVE_HEADERS,
SM_HTTP_SERVE_COOKIES,
SM_HTTP_SERVE_BODY,
SM_HTTP_SEND_FROM_CALLBACI
SM_HTTP_DISCONNECT
```

5.5. Uso de memoria y conclusiones

Lo primero que se puede observar es que con una web de apenas 270 bytes y a pesar de no haber utilizado ninguna de las funciones más avanzadas del stack, reduciendo al máximo el uso de memoria, se han consumido 25KBytes, un 76% de la memoria de programa (ROM) del PIC. Esto da una idea de lo pesado que es el stack. Será interesante observar el uso de memoria de las siguientes aplicaciones.

Por otra parte, el uso de una página web con las que transmitir variables de trabajo dentro de un entorno didáctico, resulta el acercamiento menos complejo al uso del stack. Como se ha visto, no requiere de una gestión de la conexión y solo se deben inicializar las variables de nuestra aplicación principal que queramos observar.

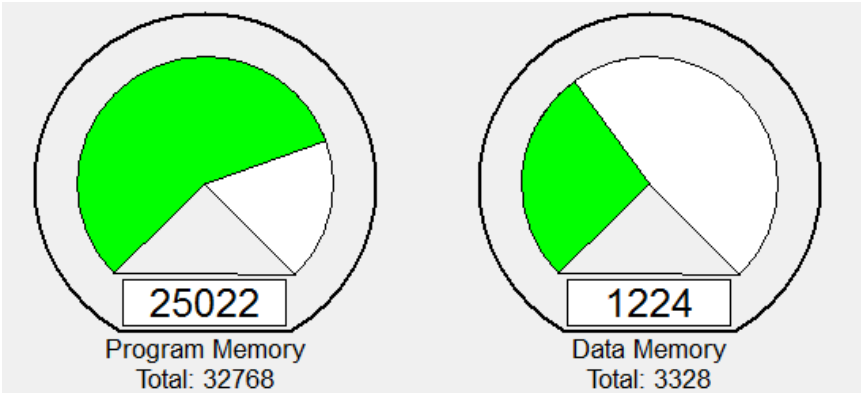


Fig. 5.9 Uso de memoria del MCU1. Fuente: *MAQ1-ServidorWeb.mcw*

6. MAQ2-Servidor UDP

6.1. Introducción.

En esta aplicación se utilizan dos microcontroladores (MCU1 y MCU2) con sus correspondientes sensores de temperatura adecuadamente conectados (véase la figura 7.1). Cada uno de ellos envía la temperatura de su sensor utilizando el protocolo de transporte UDP en modo *Broadcast*. Ello implica que todas las IPs de la red a los que estén conectados los MCU podrán acceder a estos paquetes de información.

Para la recepción de los paquetes de información en el PC se utiliza un script en Python. Este lenguaje permite la creación de sockets de comunicación y visualizar los datos recibidos en un terminal de manera sencilla.

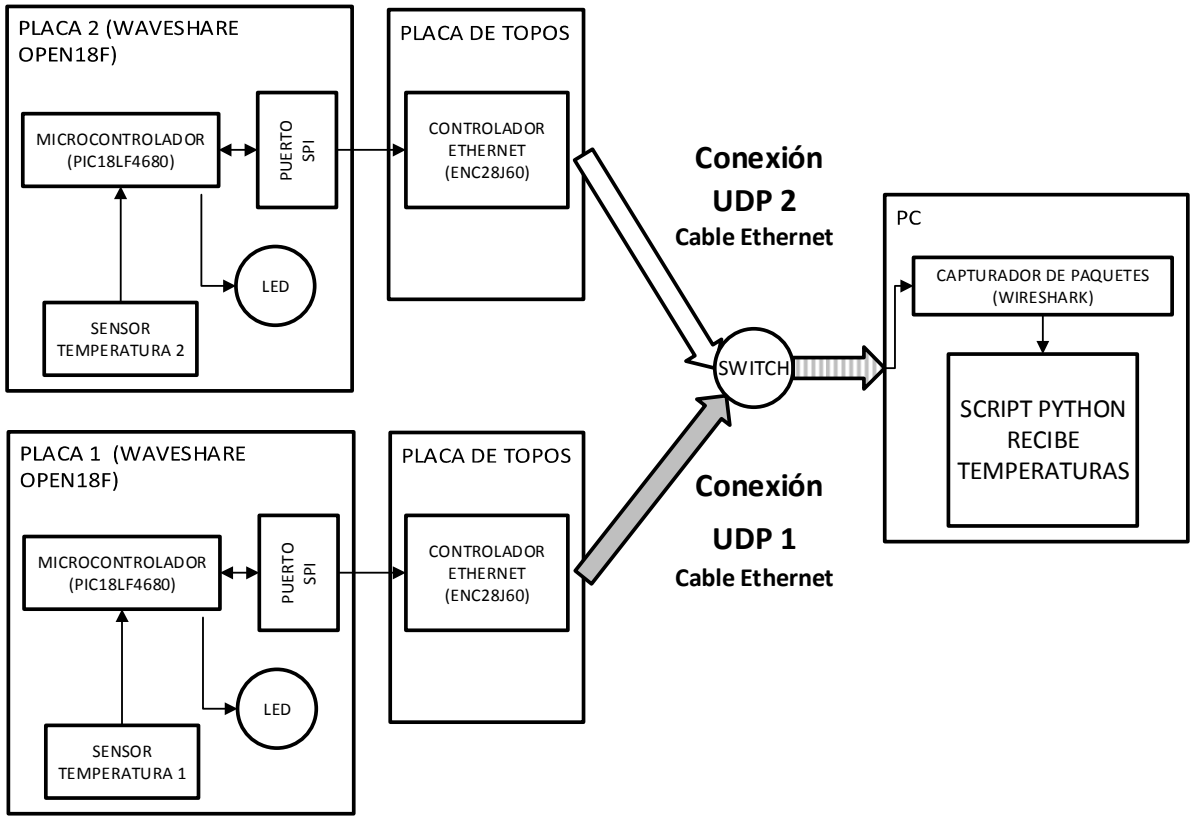


Fig. 6.1 Esquema de conexionado de MAQ2-Servidor UDP. Fuente: Propia.

Se implementa un servidor en python que escuchará en un puerto determinado e irá mostrando las temperaturas de ambas placas a medida que se vayan recibiendo los paquetes de datos.

Un número *N* de PCs que estuviesen conectados a la red y ejecutasen el script, podrían mostrar por pantalla la información enviada por los diferentes sensores.

Esta segunda aplicación sirve para validar el uso del protocolo de nivel de transporte **User Datagram Protocol (UDP** / **RFC-768)** [14], para la creación de una aplicación que envíe periódicamente la temperatura captada por un sensor a través de una red.

El protocolo UDP es uno de los dos protocolos de transporte, junto con el TCP, más utilizados, junto con el protocolo de red IP. No requiere de una confirmación de establecimiento de conexión para el envío de datos. Tampoco hay un seguimiento del mantenimiento de ésta o de la pérdida de paquetes de datos.

A cambio, es un protocolo poco pesado que introduce poca sobrecarga en la red. Es ampliamente utilizado en aplicaciones donde solo se requiera un paquete de información como respuesta, el ejemplo más claro, la obtención de una dirección IP de un servidor DNS. También es ampliamente utilizado en aplicaciones de streaming (audio, vídeo; …) donde la pérdida de un paquete de información no es crítica, debido a la gran cantidad de paquetes recibidos [14].

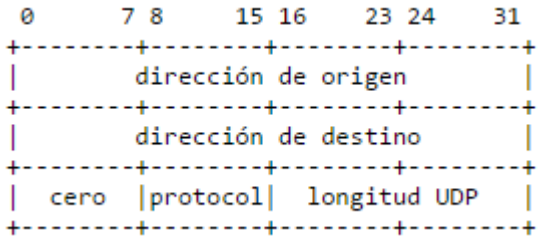


Fig. 6.2 Encabezado del protocolo IP - RFC 791 Fuente: [14].

El protocolo UDP es un protocolo de datagramas. Cada paquete contiene toda la información necesaria para ser un paquete completo y junto con el protocolo IP dispone de toda la información del remitente y el destinatario. (Fig. 6.2)

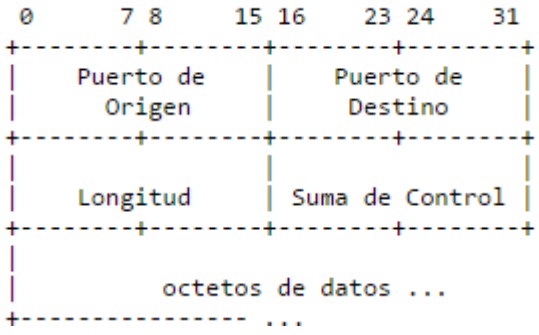


Fig. 6.3 Cabecera del protocolo UDP Fuente: [14]

Como se puede observar en la Fig. 6.3, el protocolo UDP solo requiere de una longitud y una suma de control (checksum), además de los puerto de origen y destino.

Nótese también que la información máxima a transportar por cada paquete queda limitada, por definición, a un máximo de 65507 bytes [15] (Ec. 7.1)

0xFFFF - (sizeof (encabezado IP) + sizeof (encabezado UDP)) = 65535-(20+8) = 65507 (Ec. 7.2)

6.2. Configuración del STACK.

Para esta aplicación trabajaremos directamente con el protocolo UDP y por lo tanto con la API *UDP.C*

Tampoco es necesario configurar ningún socket previamente, ya que, como se ha descrito previamente, la longitud máxima del paquete está acotada y no se mantiene ni se asegura la conexión.

MCU1

IP Address: 169.254.1.1
Gateway: 169.254.1.149
DNS1: 169.254.1.148
DNS2: 169.254.1.149

MCU2

IP Address: 169.254.1.2
Gateway: 169.254.1.149
DNS1: 169.254.1.148
DNS2: 169.254.1.149

6.3. Implementación del Script en Python.

Para esta máquina se desarrolla el siguiente script en Python

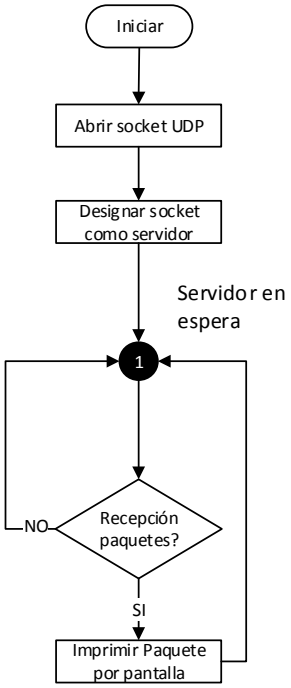


Fig. 6.4 Máquina de estados del Script Python. Fuente: *ServidorUDP.py*

```
import socket

#Configuración del puerto

port = 30303

# Apertura de un socket UDP, para ello especificando el uso del parámetro
# socket.SOCK_DGRAM informará de que se esperan paquetes del tipo DATAGRAMA

servidorUDP = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


#El servidor aceptará paquetes enviados en el puerto 30303

servidorUDP.bind(("", port))

print ("En espera de recibir temperaturas:", port)

#El servidor comprueba una vez por ciclo la llegada de paquetes. Imprime por
# pantalla los datos recibidos

while 1:
    data, addr = servidorUDP.recvfrom(1024)
    print (data)
```

Fig. 6.5 Script Python *ServidorUDP.py* Fuente: *ServidorUDP.py*

6.4. Implementación del cliente UDP.

La implementación del código se realiza en el módulo *BroadcastUDP.c.* EL flujo de datos de cualquier aplicación deberá seguir el siguiente esquema:

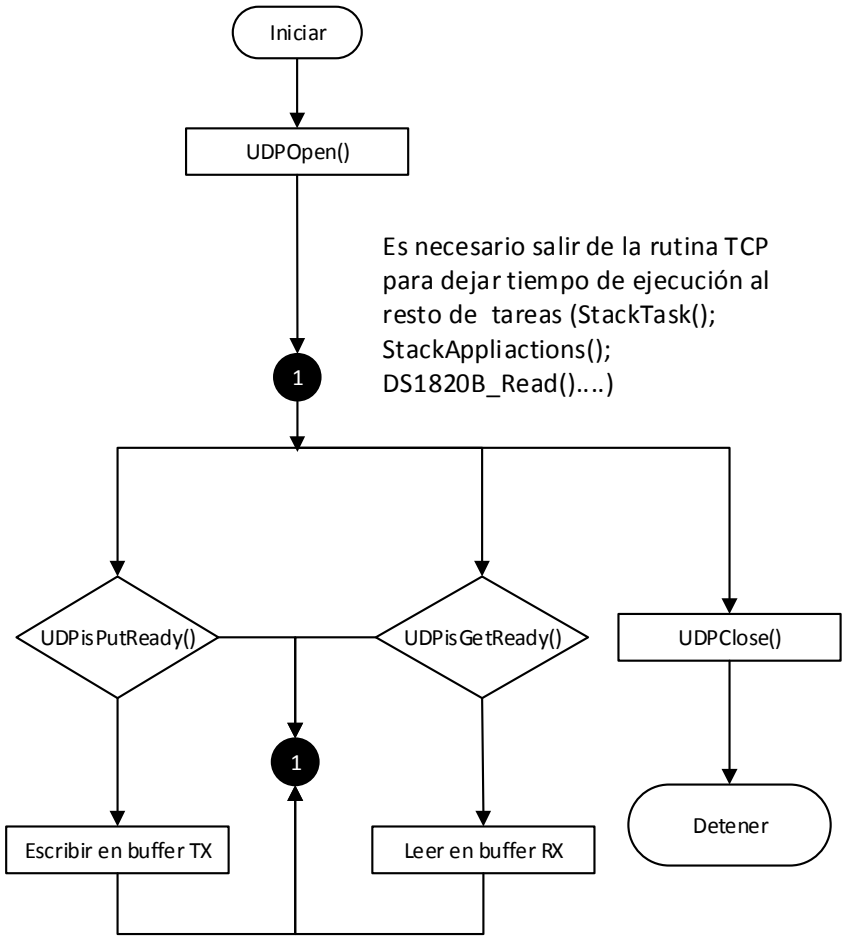


Fig. 6.6 Máquina de estados de una conexión UDP. Fuente: Propia

A continuación se muestra la configuración de parámetros de la función UDPOpen() para poder llevar acabo la aplicación.

UDP_SOCKET UDPOpen(1; 2; 3)

1) UDP_PORT localPort → *Designa el puerto local al que se asignará el socket, si se utiliza el puerto 0 se asigna uno dinámicamente.*

2) NODE_INFO* remoteNode→ *Dirección IP o MAC a donde se va a realizar la conexión, si se utiliza el parámetro NULL se entiende que se trata de un servidor o se utiliza la dirección Broadcast.*

***3)* UDP_PORT remotePort→** Puerto a donde se realiza la conexión

En este caso queda de la siguiente manera.

SocketUDPTemperatura = UDPOpen(0, NULL, PUERTO_SERVIDOR_PC);

La implementación del código se encuentra en el archivo *BroadcastUDP.c*

```
#define PUERTO_SERVIDOR_PC  30303


extern NODE_INFO remoteNode;

BYTE TEMP1[12];

void BroadcastUDP(void)
{
    UDP_SOCKET  SocketUDPTemperatura;
    BYTE        i;

    SocketUDPTemperatura = UDPOpen(0, NULL, PUERTO_SERVIDOR_PC);


    if(SocketUDPTemperatura == INVALID_UDP_SOCKET)
        return;

        if(!UDPIsPutReady(SocketUDPTemperatura))
            {
                UDPClose(SocketUDPTemperatura);
            return;
            }

    UDPPutROMString((ROM BYTE*)"\r\nLa temperatura en la estación MCU2 es de: ");
    UDPPutString(TEMP1);


    UDPFlush();


    UDPClose(SocketUDPTemperatura);
}
```

Fig. 6.7 Código BroadcastUDP.c Fuente: *Broadcast.c*

6.5. Conexión y funcionamiento.

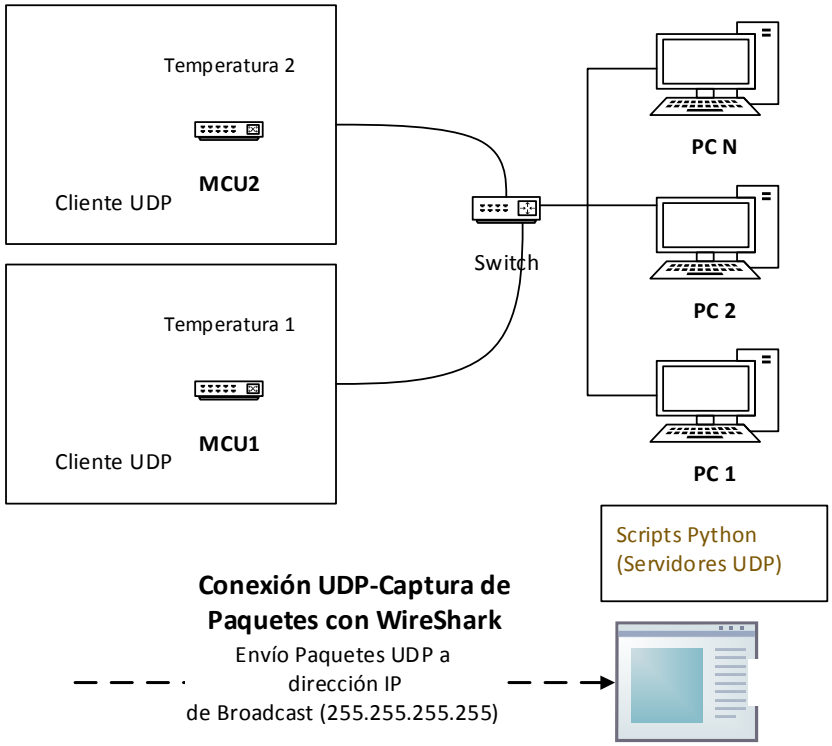


Fig. 6.8 Esquema de funcionamiento de MAQ3-Server UDP Fuente: Propia

Una vez conexionado se procede a ejecutar el script UDP Server.py y el proyecto MAQ2-UDP_Broadcast.mcw volcado en cada microcontrolador (MCU1 y MCU2)

A continuación, se muestran los paquetes captados en WireShark y guardados bajo el archivo conexion UPD.pcapng.

31	12.599685	169.254.1.2	255.255.255.255	UDP	92 4103 → 30303 Len=50
60	28.128688	169.254.1.2	255.255.255.255	UDP	92 4104 → 30303 Len=50
103	43.605358	169.254.1.2	255.255.255.255	UDP	92 4105 → 30303 Len=50
108	47.844660	169.254.1.1	255.255.255.255	UDP	92 4096 → 30303 Len=50
118	59.132322	169.254.1.2	255.255.255.255	UDP	92 4106 → 30303 Len=50
150	74.607427	169.254.1.2	255.255.255.255	UDP	92 4107 → 30303 Len=50
164	77.887921	169.254.1.1	255.255.255.255	UDP	92 4097 → 30303 Len=50
181	90.138707	169.254.1.2	255.255.255.255	UDP	92 4108 → 30303 Len=50
183	93.362924	169.254.1.1	255.255.255.255	UDP	92 4098 → 30303 Len=50
190	105.612848	169.254.1.2	255.255.255.255	UDP	92 4109 → 30303 Len=50
193	108.888862	169.254.1.1	255.255.255.255	UDP	92 4099 → 30303 Len=50

Fig. 6.9 Captura de paquetes en WireShark de la MAQ2- UDP Broadcast. Fuente: Conexion UPD.pcapng

Se puede observar como todos los paquetes van dirigidos a la dirección de broadcast 255.255.255.255 y hacia el puerto 30303. Esto implica que cualquier servidor de la red donde estén situados estas dos placas podrá recibir los paquetes, siempre que mantenga abierto un socket en el puerto correspondiente (30303).

Finalmente destacar que no existe *confirmación* ninguna por parte del PC de que haya recibido los paquetes mostrados en pantalla. De hecho, se ha realizado toda la transmisión de la información sin que el script envíe ningún mensaje de vuelta. Se debe tener en cuenta que, si bien esta característica del UDP ha facilitado su implementación en este caso, no es posible asegurar el orden correcto de los paquetes. En otro tipo de aplicaciones este podría ser un punto crítico.

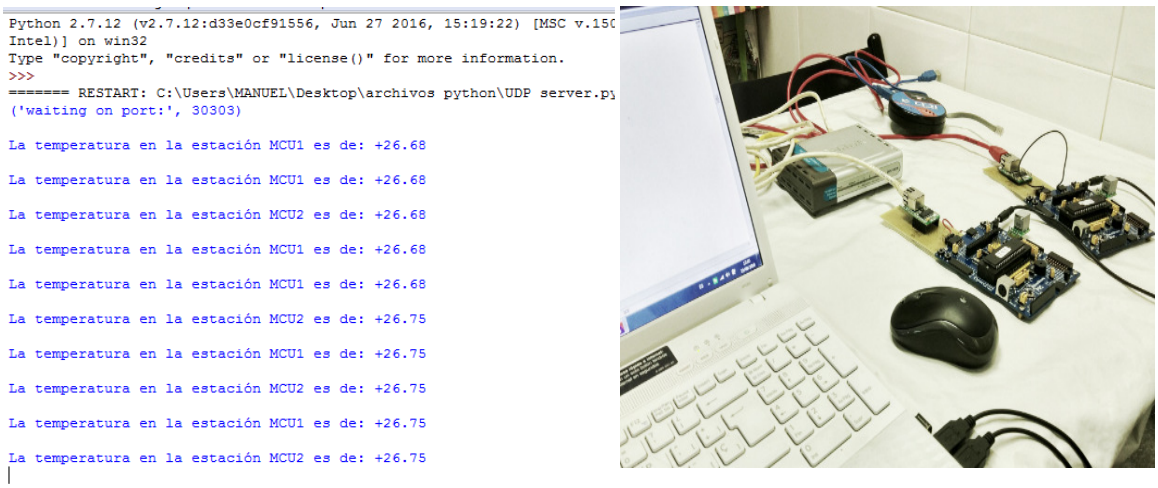


Fig. 6.10 Paquetes mostrados por el terminal en escucha. Fuente: UDP server.py

6.6. Uso de memoria y conclusiones.

Lo primero que llama la atención es la reducción del uso de memoria del stack (Fig. 6.12). Se pasa de utilizar el 76% de la memoria en la MAQ1- Servidor Web a un 36% de la memoria de programa (ROM) del PIC. Este hecho hace posible que podamos utilizar el stack para aplicaciones cuyo bucle principal (*MainDemo.c*) requiera de un mayor uso de ciclos de cálculo del PIC y de memoria de éste. Para estas aplicaciones deberemos, sin embargo, tener en cuenta dos puntos:

- La subrutina de comunicación que vaya a ser programada utilizando el protocolo UDP deberá seguir siempre una máquina de estados como la descrita en la Fig. 6.11 *Máquina de estados de una conexión UDP* .
- Debido a la naturaleza de la conexión UDP se deberá minimizar el envío de paquetes de datos y asegurar, mediante algún tipo de acuse de recibo, de aquellos paquetes que sean críticos para nuestra aplicación.

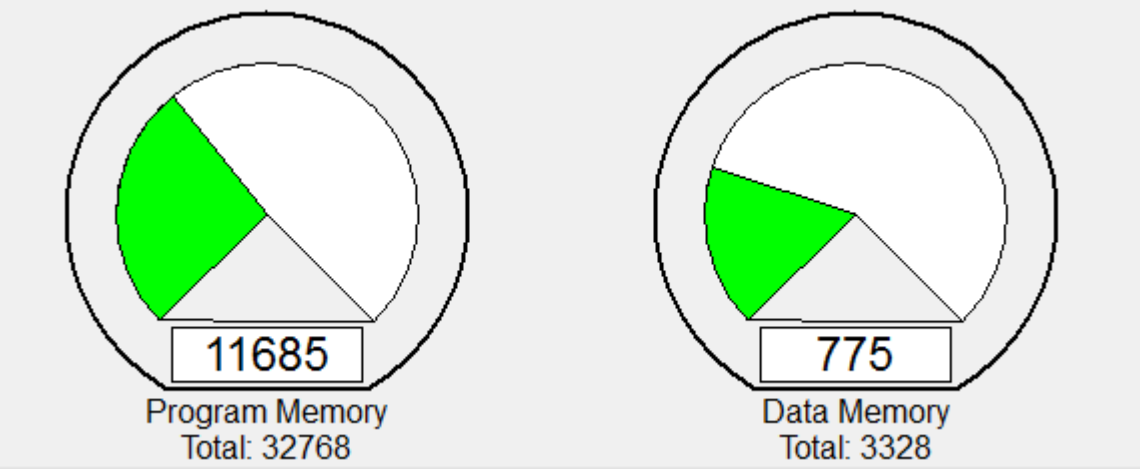


Fig. 6.12 Uso de memoria del MCU de MAQ2-ServidorUDP. Fuente: *MAQ2-UDP_Broadcast.mcw*

7. MAQ3-Servidor Web con Servidor TCP

7.1. Introducción.

En esta tercera aplicación se implementa un servidor web (MCU1) utilizando la API *HTTP.c* del stack de Microchip para que muestre la temperatura de dos sensores, situados cada uno en un microcontrolador (MCU1; MCU2) y servirlas en una única página web.

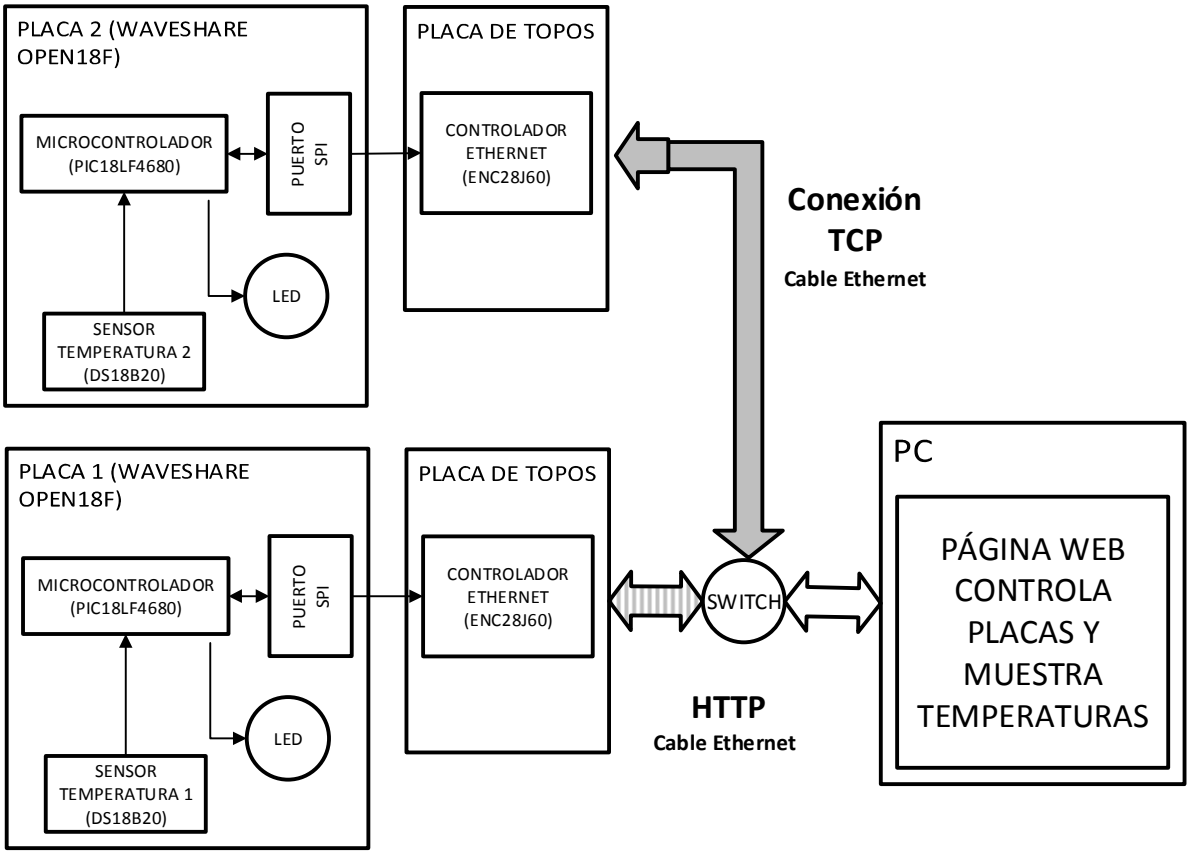


Fig. 7.1 Esquema de conexionado de MAQ3-Servidor Web con Servidor TCP Fuente: Propia

Como el servidor web MCU1 requiere de la obtención de los datos de temperatura de un segundo sensor, situado en MCU2, se debe añadir un cliente TCP/IP al MCU1 que obtenga la temperatura de un segundo microcontrolador (MCU2) y la guarde en una variable dinámica.

Además, y a diferencia de la MAQ1-Servidor Web, el usuario puede, a través del navegador, regir la actualización de las variables dinámicas que muestran la temperatura. Para ello se implementa una nueva funcionalidad de tipo GET en la aplicación HTTP.

Esta tercera aplicación servirá para estudiar la funcionalidad de una aplicación que utilice el protocolo de nivel de transporte **TCP- Transmission Control Protocol (RFC-793)** [16], implementado en el stack de Microchip.

A diferencia del protocolo UDP, el protocolo TCP requiere de:

1. Establecimiento conexión
2. Seguimiento de paquetes
3. Cierre de conexión

El protocolo TCP también garantiza el orden de todos los paquetes de datos. En caso contrario existen procedimientos de retransmisión de paquetes perdidos o erróneos.

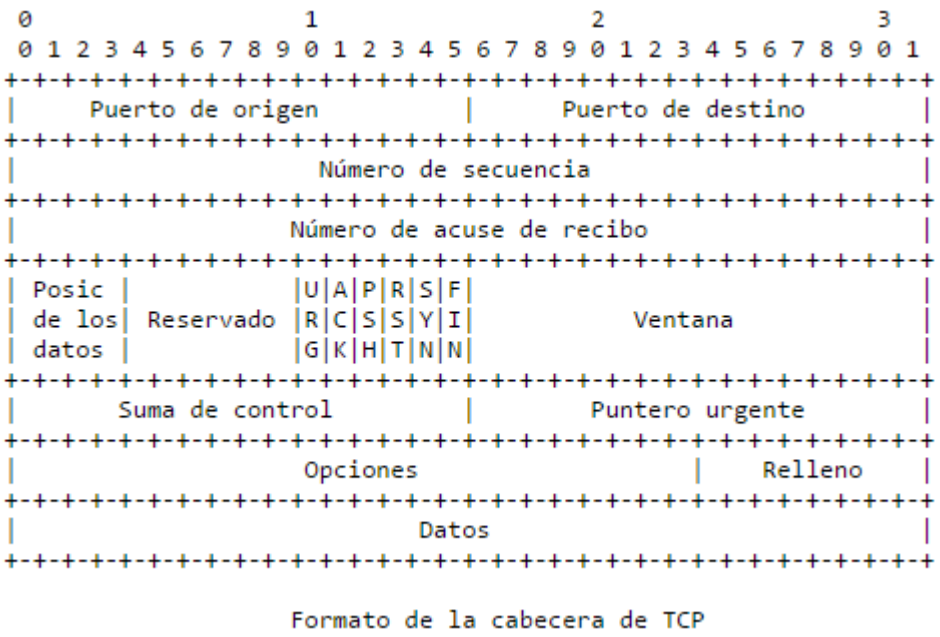


Fig. 7.2 Cabecera TCP Fuente: [16]

Nótese la diferencia en el encabezado con respecto al protocolo UDP, donde los números de secuencia (*Seq*), acuse de recibo (*Ack)* y los flags (*URG; ACK; PSH; RST; SYN; FIN)*

gobiernan gran parte del funcionamiento del protocolo, como se explica en posteriores apartados.

Debido a que no se puede monitorizar con WireShark la conexión TCP cliente-servidor entre MCU1 y MCU2, se van a proceder a escribir dos scripts en Python para realizar una conexión TCP de prueba entre las placas y el PC (Fig. 7.3), mostrando así los pasos de conexión del protocolo TCP y comprobar el correcto funcionamiento de las placas. Todo ello antes de proceder al montaje del conjunto MAQ3.

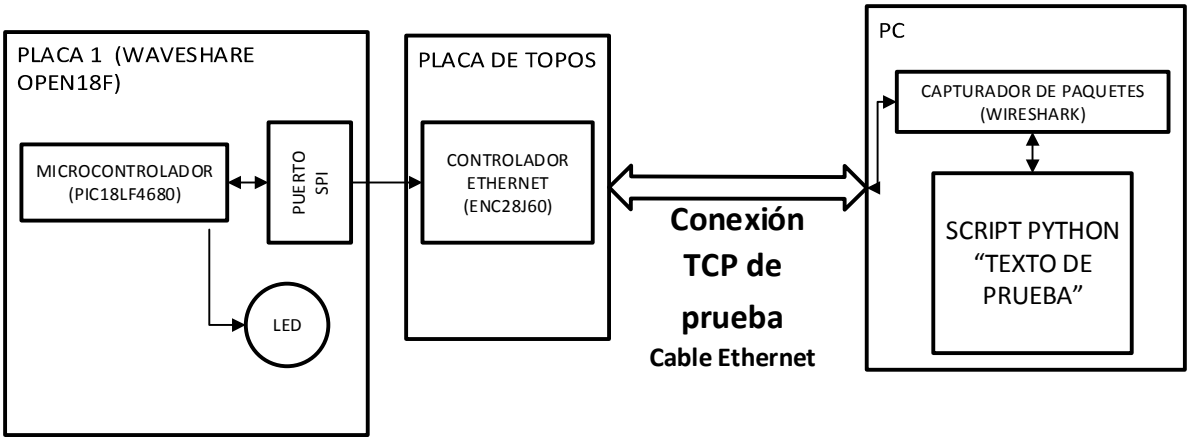


Fig. 7.3 Esquema de conexión de Servidor TCP de prueba. Fuente: Propia

7.2. Configuración del STACK.

MCU1 - Microcontrolador 1

Para este MCU1 utilizaremos la misma configuración que se ha utilizado en la MAQ1. Se trata de activar el módulo de HTTP, desactivando el resto de módulos para ahorrar memoria. Al activar el módulo de servidor HTTP, quedan activadas las API de niveles inferiores que son necesarias para la creación de un servidor HTTP (ARP, TCP y IP).

Se parte de la subrutina *GenericTCPClient.c* para crear la máquina de estados, que seguirá el algoritmo cliente.

Es necesaria la configuración de dos sockets de conexión. El primero para el servidor Web y el segundo para el cliente TCP.

```
 #define TCP_PURPOSE_HTTP_SERVER 8
{TCP_PURPOSE_HTTP_SERVER, TCP_ETH_RAM, 500, 500},
 #define TCP_PURPOSE_GENERIC_TCP_CLIENT 0
{TCP_PURPOSE_GENERIC_TCP_CLIENT, TCP_ETH_RAM, 200, 200},
```

Fig. 7.4 Sockets de conexión. Fuente: *TCPConfig.h*

IP Address: 169.254.1.1
Gateway: 169.254.1.148

La dirección IP Gateway suele ser la dirección IP de un router de la red en la que nos encontremos, ya que el router dispone de la tabla de direcciones MAC de todas las IPs de la red. En este caso se utiliza una de las direcciones IP a la que está conectado físicamente el MCU.

MCU2 - Microcontrolador 2

Con este MCU trabajaremos directamente con el protocolo TCP y por lo tanto, con la API *TCP.c*., se parte de la subrutina *GenericTCPServer.c* para crear la máquina de estados que seguirá el algoritmo.

Es necesaria la configuración de un socket de conexión.

```
#define TCP_PURPOSE_GENERIC_TCP_SERVER 1
 {TCP_PURPOSE_GENERIC_TCP_SERVER, TCP_ETH_RAM, 200, 200},
```

IP Address: 169.254.1.2
Gateway: 169.254.1.148

7.3. Implementación de los Scripts Python en PC.

Como se ha descrito en la introducción, antes de la conectar MCU1 y MCU2 entre sí, se procede a la creación de dos scripts en Python con los que visualizar, mediante Wireshark, la correcta ejecución de la máquina de estados de ambos microcontroladores. Con ello se verifica el correcto funcionamiento de las subrutinas *GenericTCPClient.c y GenericTCPServer.c.*

Script Servidor TCP en PC- Cliente TCP en MCU1

El script servidor quedará a la espera de una conexión e imprimirá por pantalla los datos que envíe el MCU1, que hará de cliente. Una vez recibidos los datos, el script los enviará de vuelta y después procederá a desconectar.

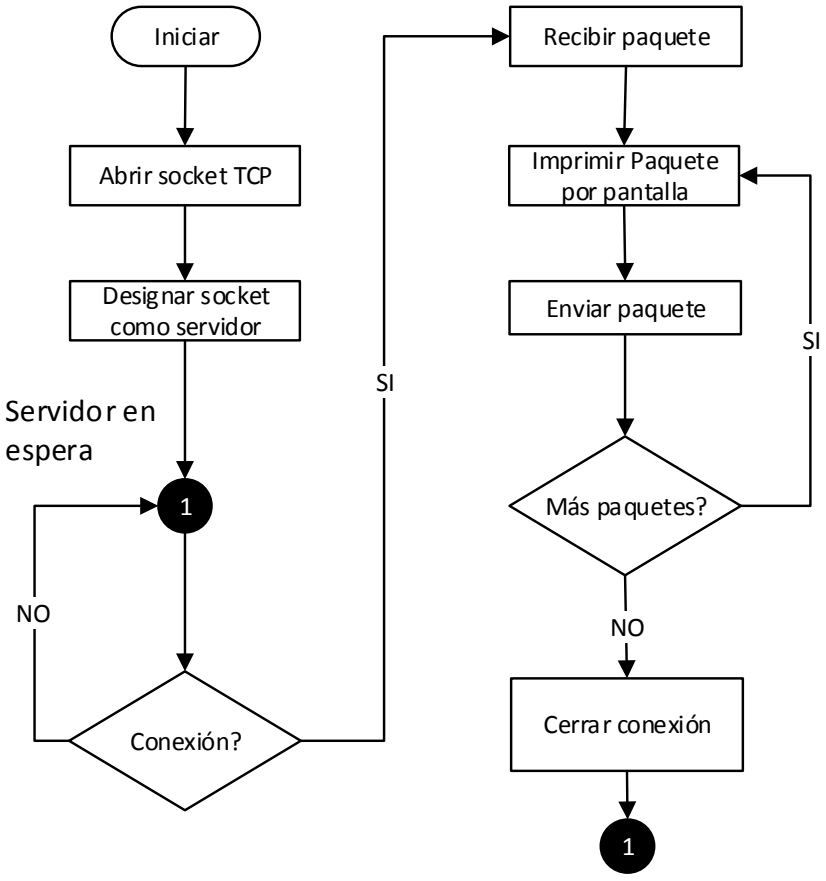


Fig. 7.5 Máquina de estados del servidor TCP de prueba escrito en Python Fuente: Propia

```
import socket
import sys

#En este caso el PC utilizado es 'MANUEL-VAIO' pero utilizando
# la funcion gethostbyname obtendríamos el mismo resultado.
#Muestra la IP del PC.

for host in [ 'MANUEL-VAIO' ]:
    try:
        print'%15s : %s' % (host, socket.gethostbyname(host))
    except socket.error, msg:
        print '%15s : ERROR: %s' % (host, msg)

#El parámetro SOCK_STREAM identifica al socket como TCP

serversocket= socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to the port
print(socket.gethostname())


server_address = ((socket.gethostname(), 1060))
print (sys.stderr, 'Inicializa socket en %s puerto %s' % server_address)
serversocket.bind(server_address)

# Convertimos el socket en un servidor. Estará esperando la entrada
#de datos en el buffer RX
serversocket.listen(1)



while True:
    # Espera e inicio del bucle principal
    print ('A la espera de conexión')
    connection, client_address = sock.accept()

    try:
        print (sys.stderr, 'Conexión desde', client_address)

        # Recibe datos en trozos de 16 bytes
        while True:
            datos = connection.recv(16)
            print (sys.stderr, 'recibidos "%s"' % datos)
            if datos:
                print (sys.stderr, 'enviando de vuelta')
                connection.sendall(datos)
                print (datos)
            else:
                print (sys.stderr, 'no hay más datos')
                 print (sys.stderr, 'cierre de conexión', client_address)
                connection.close()
                break
```

```
finally:
    # Cierre
    connection.close()
```

Fig. 7.6 Script Python *ServerTCP.py.* Fuente: *ServerTCP.py*

Script Cliente TCP en PC- Servidor TCP en MCU1

El script clientes intentará establecer una conexión, enviará un texto y quedará a la espera de que sea devuelto. Después procederá a desconectar.

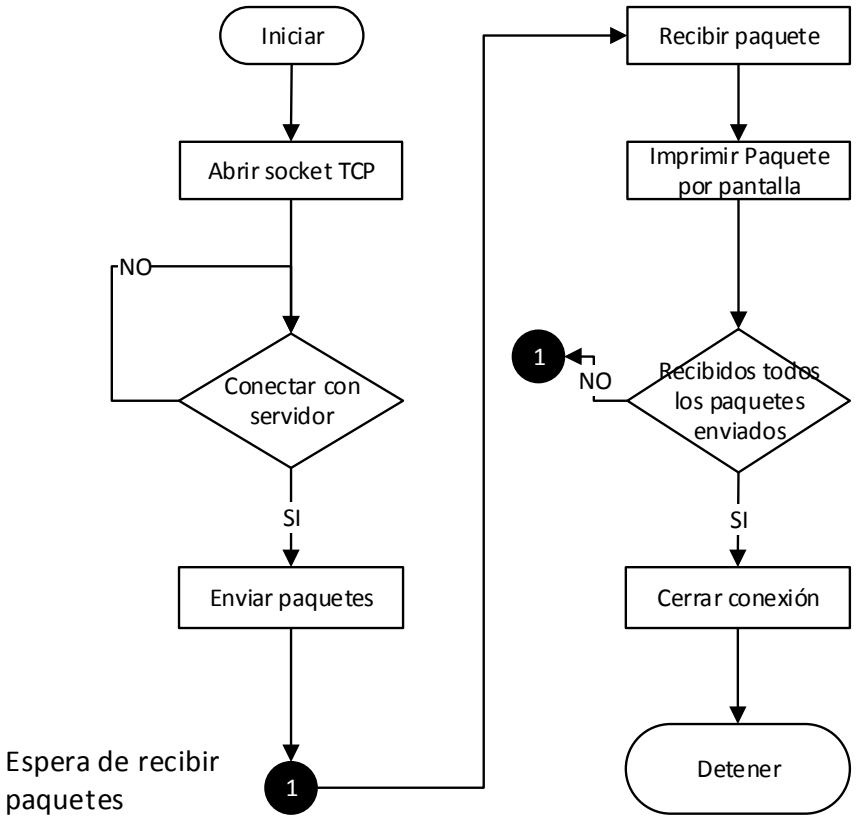


Fig. 7.7 Máquina de estados del Script en Python *ClienteTCP.py*. Fuente: Propia

```
server_address = (host, port)
print (sys.stderr, 'Conectando %s puerto %s' % server_address)
sock.connect(server_address)

try:

    # Envio del texot
    message = 'espero el texto en mayusculas.'
    print (sys.stderr, 'enviando "%s"' % message)
    sock.sendall(message)


    # Espera de respuesta
    amount_received = 0
    amount_expected = len(message)

    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print (sys.stderr, 'recibido "%s"' % data)

finally:

    sock.close()
```

Fig. 7.8 Script Python *ClienteTCP.py.* Fuente: *Cliente TCP.py*

7.4. Implementación de Cliente y Servidor TCP (MCU1 y MCU2)

Una vez desarrollados los scripts en Python debe implementarse el código en los microcontroladores. Para ello revisaremos la máquina de estados que necesitaremos construir en los archivos *GenericTCPClient.c* y *GenericTCPServer.c,* cuyo flujo de datos debe de seguir el siguiente esquema.

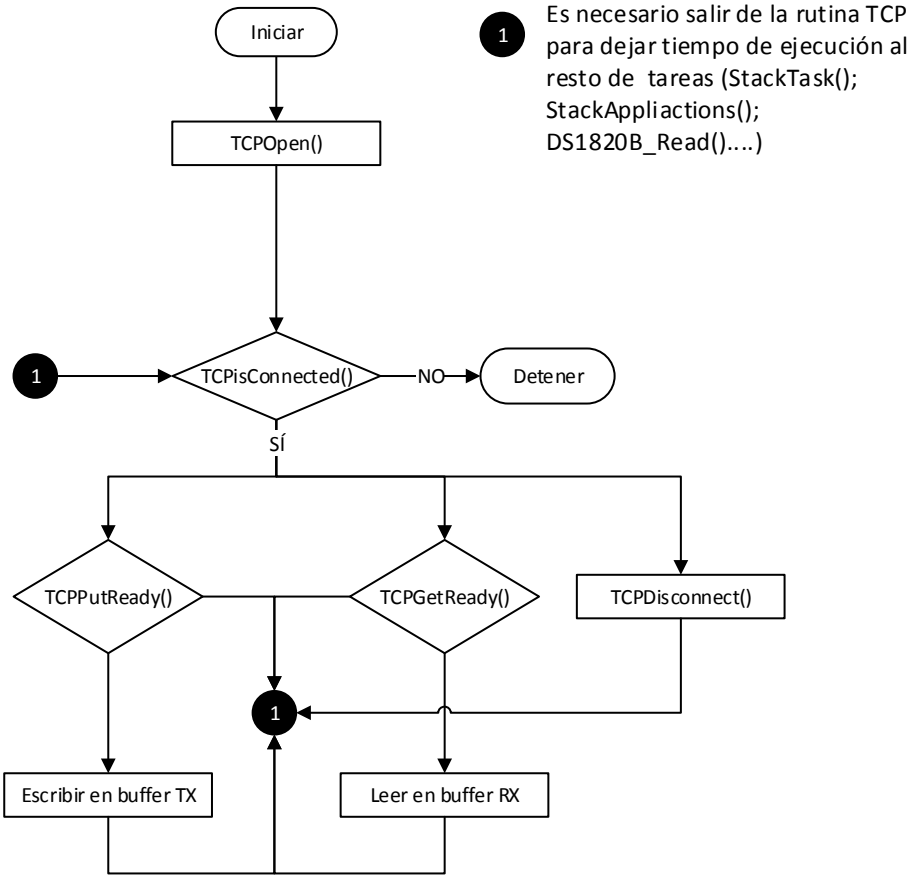


Fig. 7.9 Máquina de estados de una conexión UDP Fuente: Propia *ClienteTCP.py*. Fuente:

Si bien el flujo de datos es muy parecido al mostrado en *BroadcastUDP.c,* deben tenerse en cuenta las diferencias de funcionamiento del protocolo.

Véase como ejemplo el uso de *TCPOpen()*, cómo está implementada esta función en el stack y cómo pone en marcha una segunda máquina de estado que no existe en el protocolo UDP.

TCP_SOCKET TCPOpen(1; 2; 3;4)

1) dwRemoteHost →Para aquellos sockets de tipo cliente. El parámetro de entrada dwRemoteHost es el puntero que designa el host remoto, en formato string ("www.etseib.upc.edu", "192.168.1.123") o en forma hexadecimal (0x7B01A8C). Como no se resolverá una dirección DNS, en este caso introduciremos la dirección IP en forma literal hexadecimal.

2) vRemoteHostType→ Según el tipo de socket TCP que se abra, disponemos de cinco opciones:

• TCP_OPEN_SERVER
Abre el socket como servidor, ignora el parámetro dwRemoteHost.
• TCP_OPEN_RAM_HOST
Abre una dirección IP especificada por una STRING. Require de DNS.
• TCP_OPEN_ROM_HOST
Igual que el anterior pero el puntero es guardado en ROM.
• TCP_OPENIP_ADDRESS
Abre una dirección IP especificada por un literal.
• TCP_OPEN_NODE_INFO
Abre un socket de tipo cliente a un host especificado por un nodo.

3) WORD wPort,→ El puerto en el que se abrirá el socket.

4) BYTE vSocketPurpose→Cualquiera de las constantes TCP_PURPOSE definidas en el archivo de configuración *TCPIPConfig.h*

TCPOpen (dHostIP, TCP_OPEN_IP_ADDRESS, ServerPort, TCP_PURPOSE_GENERIC_TCP_CLIENT);

Esta función pone en marcha la máquina de estados TCP STATE del archivo *TCP.c.* Para analizar el funcionamiento del protocolo, se va a proceder a analizar cómo se ha implementado la máquina de estados en el stack de Microchip que regula su funcionamiento.

Análisis de la máquina de estados TCP STATE

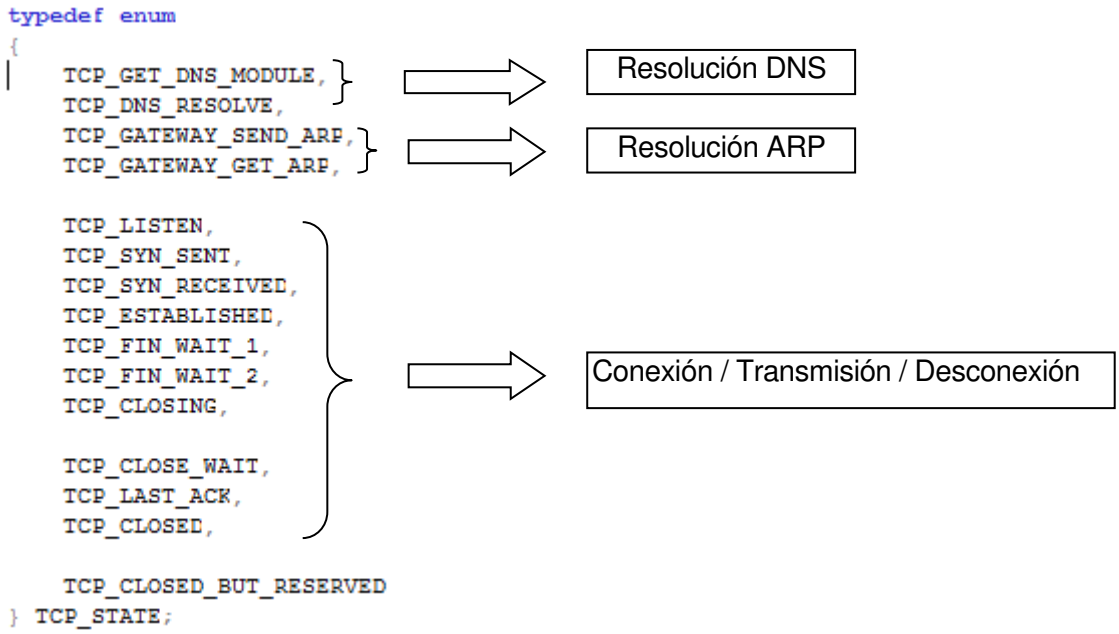


Fig. 7.10. Estados TCP STATE. Fuente: *TCP.c*

En la Fig. 7.10 podemos observar los estados con los que ha implementado Microchip el protocolo TCP. Para facilitar su comprensión, se dividen los estados de TCP STATE en tres partes.

Resolución DNS

La resolución DNS realiza una llamada al puerto 53 de un servidor predeterminado en configuración *TCPIPConfig.h*, para que resuelva el dominio indicado en el parámetro **dwRemoteHost** de la función TCP Open. El servidor DNS resuelve el dominio enviando en un encapsulado UDP la dirección IP a la que corresponde ese dominio. Todo ello se realiza en otra máquina de estados descrita en *DNS.C*

Esta máquina no utiliza el proceso de resolución de DNS. Las direcciones IP que utilizan todas las aplicaciones son proporcionadas directamente en forma de literal dentro del código *GenericTCPServer.c.*

```
// State machine for a DNS query
static enum
{
    DNS_START = 0,              // Initial state to reset client state
    DNS_ARP_START_RESOLVE,      // Send ARP resolution of DNS server
    DNS_ARP_RESOLVE,            // Wait for response to ARP request
    DNS_OPEN_SOCKET,            // Open UDP socket
    DNS_QUERY,                  // Send DNS query to DNS server
    DNS_GET_RESULT,             // Wait for response from DNS server
    DNS_FAIL,                   // ARP or DNS server not responding
    DNS_DONE                    // DNS query is finished
} smDNS = DNS_DONE;
```

Fig. 7.11 Estados de DNS query. Fuente: *DNS.c*

Resolución ARP

Estos dos estados de la Fig. 7.10 hacen referencia al protocolo de mediación **Address Resolution Protocol** basado en el estándar **(RFC 826)** [18] y no al protocolo TCP en sí mismo, aunque como se verá es necesario la resolución del ARP para poder establecer una conexión.

El protocolo ARP se encarga de resolver / relacionar direcciones de protocolos de redes, en nuestro caso las direcciones IP de una red, con las direcciones físicas de las máquinas que la constituyen.

La dirección física de una red es la denominada MAC (Media Access Control). Sin entrar en detalles, las direcciones MAC son direcciones globales únicas de 6 bytes insertadas en las tarjetas Ethernet, en el momento de su fabricación. Como por ejemplo la dirección 00:02:AF:1B:2F:01 [17].

En este caso y dentro de una red IPv4, el funcionamiento del protocolo es el siguiente.[19]

- Se envía un mensaje a una dirección MAC de broadcast FF:FF:FF:FF:FF:FF a todos los nodos a lo que esté conectado la máquina en cuestión. El mensaje contiene en su interior la dirección que se está buscando.

- Si el nodo que recibe el mensaje no es o no dispone en su cache la dirección IP que se especifica en el mensaje, repite el mensaje al resto de nodos a los que esté conectado. Una vez un nodo encuentra en su cache de direcciones la IP que se busca, envía la dirección MAC.

Conexión / Transmisión / Desconexión TCP

Los estados restantes de la máquina TCPState del módulo *TCP.c* corresponden con los estados descritos en el estándar **Transport Control Protocol (RFC-793)** [16] **[20]** y su definición:

LISTEN - representa la espera de una solicitud de conexión proveniente de cualquier TCP y puertos remotos.

SYN-SENT - Representa la espera de una solicitud de conexión concordante tras haber enviado previamente una solicitud de conexión.

SYN-RECEIVED - Representa la espera del acuse de recibo confirmando la solicitud de conexión tras haber recibido tanto como enviado una solicitud de conexión.

ESTABLISHED - Representa una conexión abierta, los datos recibidos pueden ser entregados al usuario. El estado normal para la fase de transferencia de una conexión.

FIN-WAIT-1 - Representa la espera de una solicitud de finalización de la conexión proveniente del TCP remoto, o del acuse de recibo de la solicitud de finalización previamente enviada.

FIN-WAIT-2 - representa la espera de una solicitud de finalización del TCP remoto.

CLOSE-WAIT - Representa la espera de una solicitud de finalización de la conexión proveniente del usuario local.

CLOSING - representa la espera del paquete, proveniente del TCP remoto, con el acuse de recibo de la solicitud de finalización.

LAST-ACK - Representa la espera del acuse de recibo de la solicitud de finalización de la conexión previamente enviada al TCP remoto (lo que incluye el haber enviado el acuse de recibo de la solicitud

Fig. 7.12 Definición de los estados TCP Fuente: [20]

Este concepto de gestión de todos los pasos de conexión / transmisión / desconexión es lo que diferencia este protocolo del UDP.
A continuación, se presenta el funcionamiento teórico del protocolo.[21];[22]

- **CONEXIÓN**

El procedimiento de conexionado se realiza mediante un *handshake* o confirmación a tres bandas. Esta consiste en el envío de paquetes de información en los que se incluyen un bit de bandera (flag) dentro del encapsulado y unos números *Seq* y *Ack* que cumplen los siguientes requerimientos:

1) Envío de un paquete con el flag SYN activado y con número de secuencia aleatorio **Seq=x**.
2) Envío un paquete con los flags SYN y ACK activados. Se incluye un número de acuse de recibo (*acknowledgement*) **Ack=x+1**. Este corresponde con el número **Seq=x** enviado en el paso anterior más 1 y numero de secuencia aleatorio **Seq=y**.
3) Envío del flag ACK con el número de acuse de recibo **Ack=y+1** y el número de secuencia **Seq=x+1**.

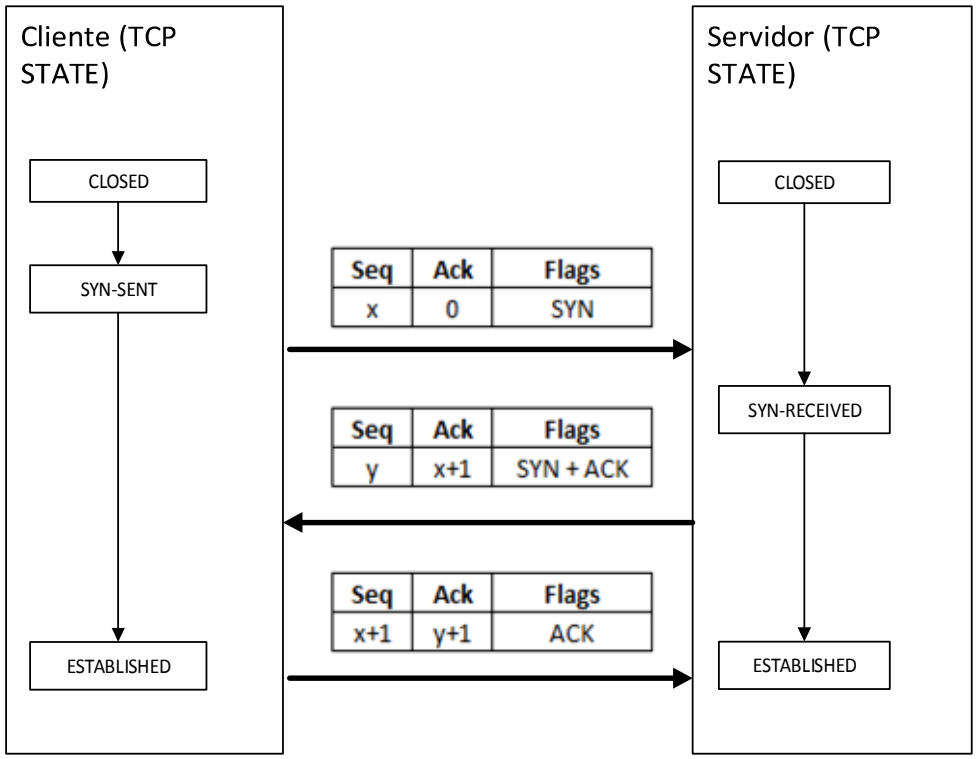


Fig. 7.13 Esquema inicio conexión TCP. Fuente: Propia

- **TRANSMISIÓN DE DATOS**

Una vez inicializada la conexión, la gestión de la correcta recepción de paquetes se realiza con los números de secuencia (*Seq*), el número de acuse de recibo (Ack), la longitud de los datos enviados (*Len*) y la disponibilidad de buffer RX de lectura que tiene el receptor para recibir datos (*Win*).

El número ***Seq*** es un contador secuencial de la cantidad de datos enviados.

El número ***Ack*** es un contador secuencial de la cantidad de datos recibidos.

Ambos son números de 4 bytes inicializados aleatoriamente, pero para facilitar la explicación establecemos que los números *Ack* y *Seq* para cliente y servidor han quedado inicializados a 1 después de la conexión (x=0; y=0). La cantidad de datos que queremos enviar es 2L y el buffer de recepción RX para cliente y servidor es 2L.

1) Envío de un paquete de información con **Len=L**, **Seq=1; Ack=1**; **Win=2L**.
2) Envió de un paquete con flag ACK; **Len=0**; **Seq=1; Ack=1+L**; **Win=2L-L**.
3) Envío de un paquete con **Len=L**; **Seq=1+L**; **Ack=1**; **Win=2L**.
4) Envío de un paquete con flag ACK; **Len=0**; **Seq=1**; **Ack=1+2L**; **Win=2L-2L**. En este último paso el servido ha quedado con el buffer de recepción lleno, por lo que el cliente no podrá continuar enviando datos hasta que el servidor vacíe el buffer RX. Una vez eso ocurra el servido enviará otro mensaje de control informando del nuevo **Win=2L**.
   Debemos tener en cuenta que el servidor podría haber enviado datos, ya que el buffer del cliente RX siempre ha estado libre.

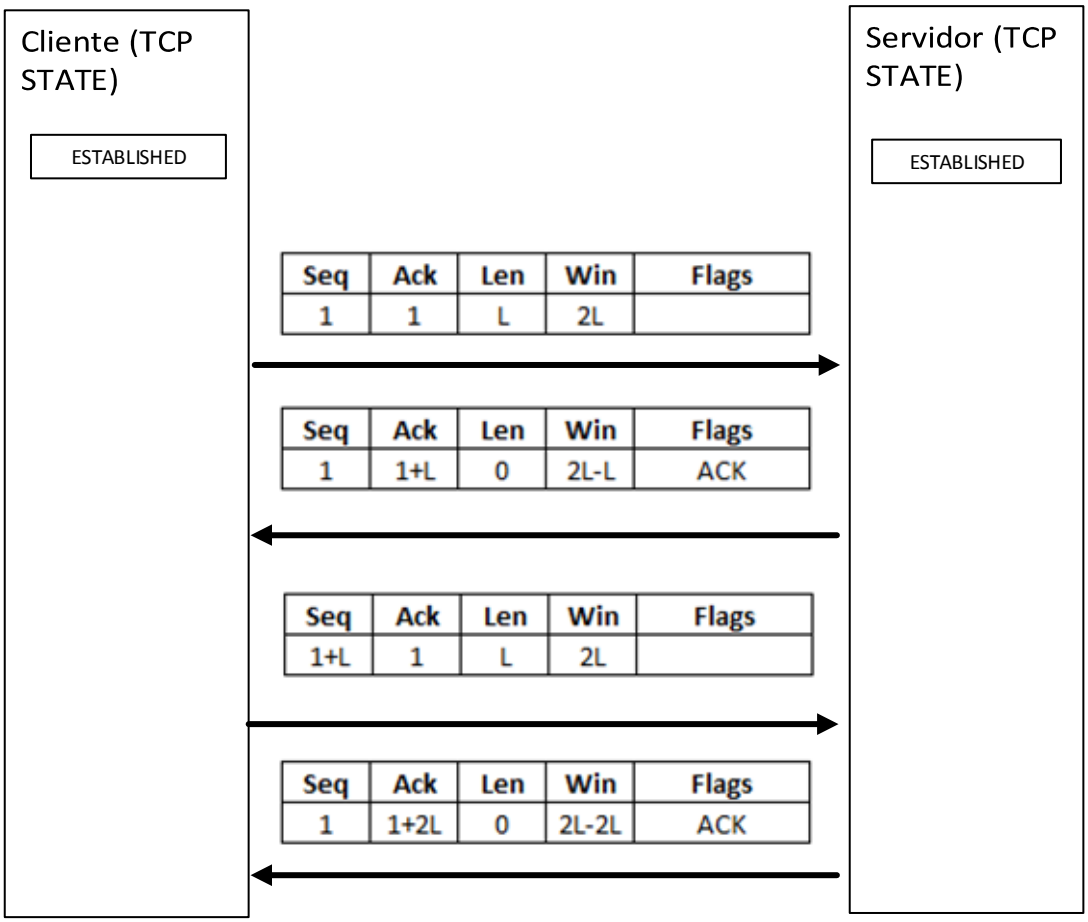


Fig. 7.14 Esquema de transmisión de datos en una conexión TCP Fuente: Propia

- **DESCONEXIÓN**

Para finalizar la conexión se procede a una desconexión con confirmación a 4 bandas, procedimiento muy parecido al del conexionado. Para ello cada punto de la conexión cliente-servidor, deberá enviar una solicitud de finalización de conexión mediante el flag FIN y confirmarla mediante el flag ACK. Continuando con el ejemplo anterior queda de la siguiente manera.

1) Envío de un paquete con el flag FIN activado y con número de secuencia **seq=1+2L; len=0; ack=1**;
2) Envío de un paquete con flag ACK; **len=0; seq=1; ack=1+2L+1; win=2L**.
3) Una vez el servidor ha acabado, se envía un paquete con el flag FIN activados **len=0; seq=1; ack=1+2L+1; win=2L**.
4) Envío de un paquete con flag ACK; **len=0; seq=1+2L; ack=1+1; win=2L.**

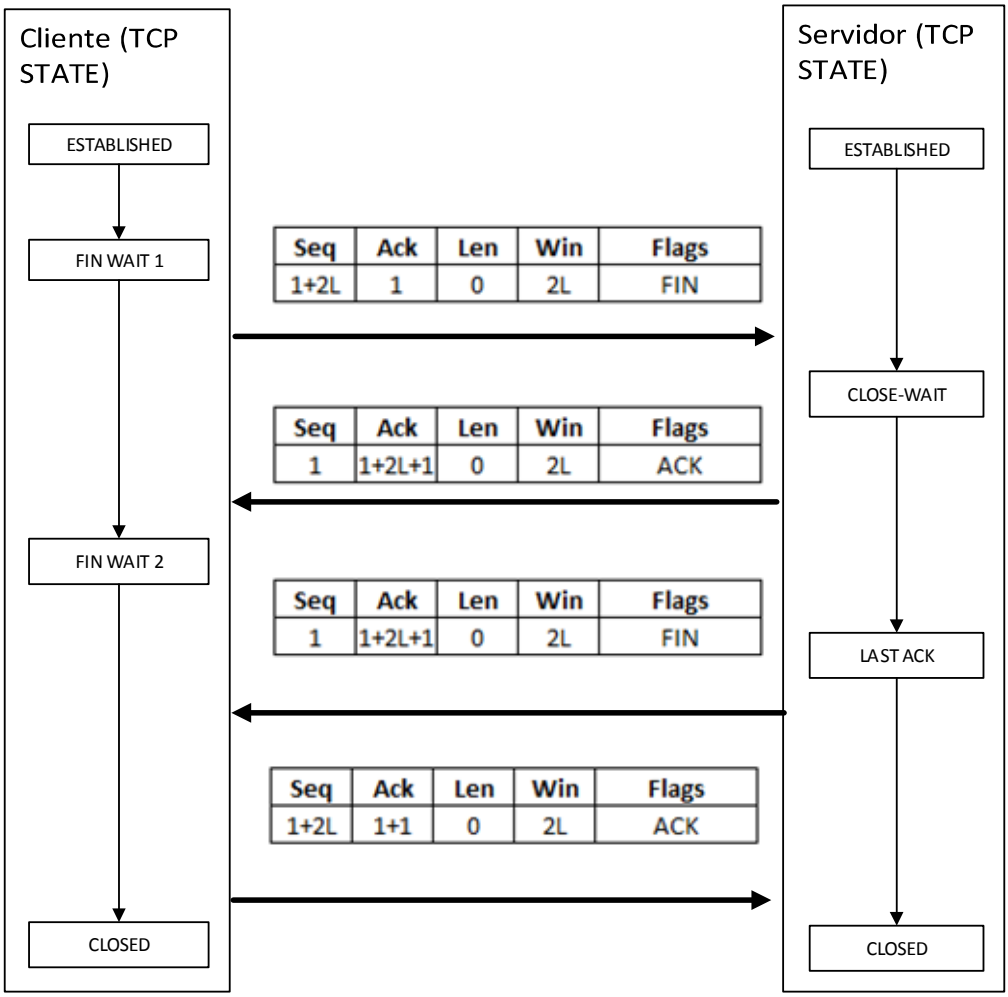


Fig. 7.15 Esquema finalización conexión TCP. Fuente: Propia

7.5. Conexión de prueba y análisis en WireShark

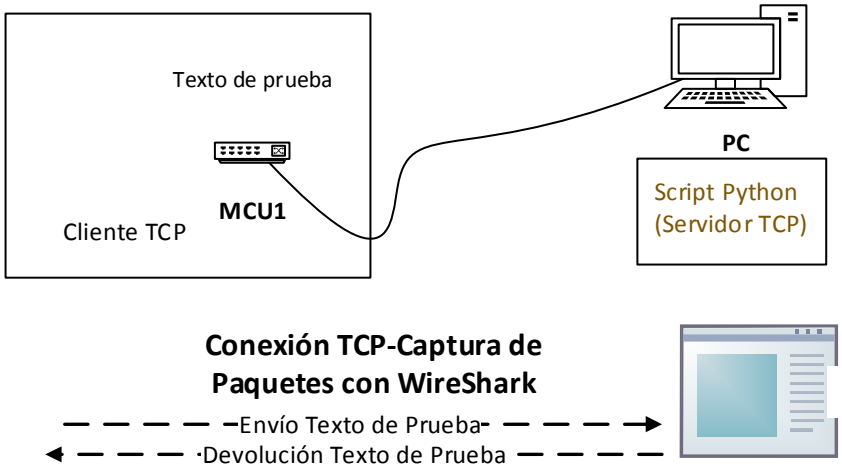


Fig. 7.16 Esquema de funcionamiento de la conexión TCP de prueba. Fuente: Propia

Una vez hecho el conexionado tal y como se indica en el esquema Fig. 8.16, se procede a ejecutar el script TCP Server.py y el proyecto MAQ3-Prueba Cliente TCP.mcw volcado en cada microcontrolador (MCU1).

A continuación, se verifica el correcto funcionamiento del cliente TCP (MCU1) conectándolo al script escrito en Python. Se monitoriza la conexión mediante WireShark y se analizan los paquetes.

Al poner en marcha el script, éste queda a la espera:

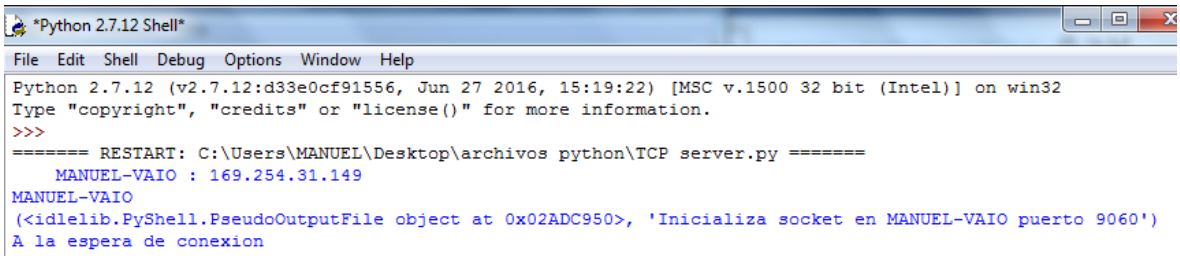


Fig. 7.17 Script Python de servidor TCP en espera. Fuente: TCP Server.py

Pasado un tiempo determinado en el bucle *MainDemo.c* con la variable *numero* que hace la función de simple contador, se procede a la ejecución de la subrutina *GenericTCPClient.c.* De esta manera se evita el envío masivo de mensajes al servidor Python.

En esta subrutina se establecerá y gestionará la conexión. Una vez establecida se enviará el siguiente mensaje *Prueba de conex.*

```
TCPPutROMString(MySocket, (ROM BYTE*)" Prueba de conex.");
```

Cuando el PC haya recibido el mensaje, éste será devuelto al MCU1, donde se guardará en una variable de buffer (*vBuffer)*.

```
case SM_PROCESS_RESPONSE:

    if(!TCPIsConnected(MySocket))
    {
        GenericTCPExampleState = SM_DISCONNECT;

    }


    w = TCPIsGetReady(MySocket);

    i = sizeof(vBuffer)-1;
    vBuffer[i] = '\0';
    while(w)
    {
        if(w < i)
        {
            i = w;
            vBuffer[i] = '\0';
        }
        w -= TCPGetArray(MySocket, vBuffer, i);

        if(GenericTCPExampleState == SM_PROCESS_RESPONSE)
            break;
    }
```

Fig. 7.18 Uso de variable vBuffer para guardar datos recibidos. Fuente: GenericTCPClient.c

Una vez enviado de vuelta el mensaje se procede a cerrar la conexión. El script quedará a la espera de una nueva conexión.

El resultado final queda mostrado por la pantalla del terminal es el siguiente:

```
File  Edit  Shell  Debug  Options  Window  Help
Python 2.7.12 (v2.7.12:d33e0cf91556, Jun 27 2016, 15:19:22) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
======= RESTART: C:\Users\MANUEL\Desktop\archivos python\TCP server.py =======
    MANUEL-VAIO : 169.254.31.149
MANUEL-VAIO
(<idlelib.PyShell.PseudoOutputFile object at 0x02BCC950>, 'Inicializa socket en MANUEL-VAIO puerto 9764')
A la espera de conexion
(<idlelib.PyShell.PseudoOutputFile object at 0x02BCC950>, 'Conexion desde', ('169.254.1.1', 1109))
(<idlelib.PyShell.PseudoOutputFile object at 0x02BCC950>, 'recibidos "Prueba de conex."')
(<idlelib.PyShell.PseudoOutputFile object at 0x02BCC950>, 'enviando de vuelta')
Prueba de conex.
```

Ahora, si observamos los paquetes de datos que ha capturado WireShark, obtenemos el siguiente flujo de datos que pasamos a analizar.

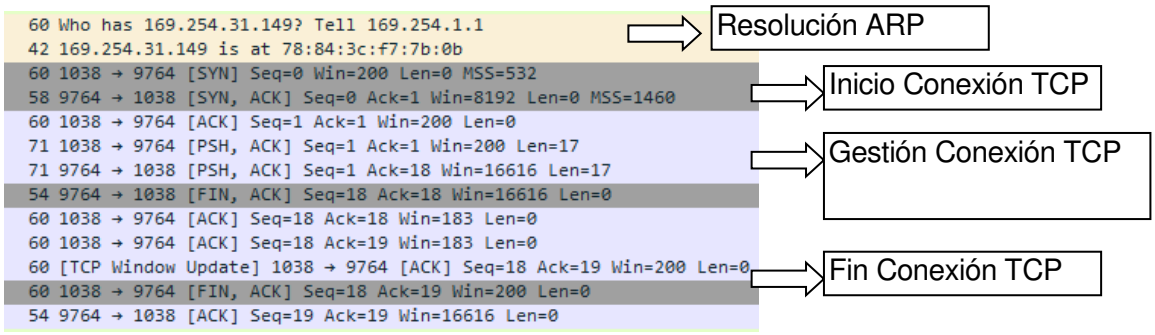


Fig. 7.19 Paquetes capturados entre MCU1 y Servidor TCP Fuente: *ConexionTCP.pcapng*

Análisis de los paquetes - Resolución ARP

Como se ha podido ver en la Fig. 7.10 que muestra los estados del stack para la gestión una conexión TCP. El primer paso es la resolución del ARP.

```
Microchi_00:00:00    Broadcast          ARP     60 Who has 169.254.31.149? Tell 169.254.1.1
SonyCorp_f7:7b:0b    Microchi_00:00:00 ARP     42 169.254.31.149 is at 78:84:3c:f7:7b:0b
```

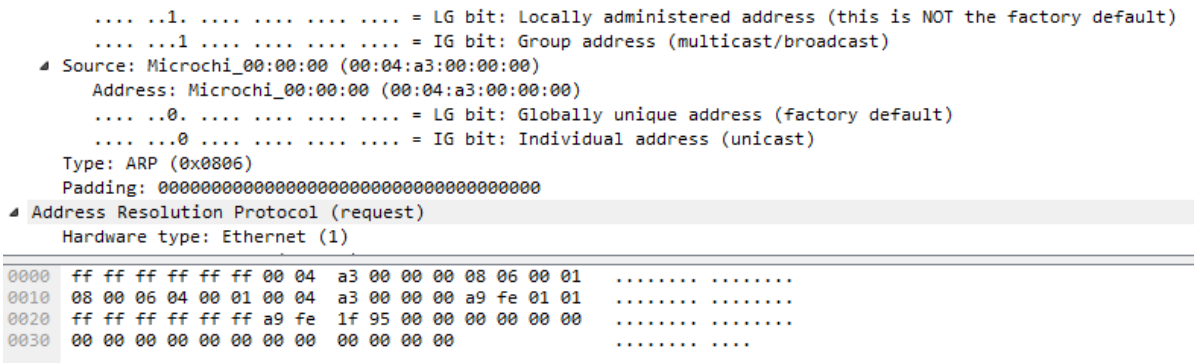


Fig. 7.20 Paquete ARP caputaro con Wireshark Fuente: *Conexion TCP.pcapng*

Si analizamos el paquete, vemos que la dirección MAC a la que se envía el mensaje es la dirección de broadcast (FF.FF.FF.FF.FF) y que se está buscando la dirección IP hexadecimal A9.FE.1F.95 (169.254.31.149).

Seguidamente en la Fig. 7.10 encontramos la conexión al servidor creado en el PC. Aquí se muestran los pasos de *handshake* o establecimiento de la conexión y lo pasos por los distintos estados de la máquina de estados definida en el código TCP.c

Análisis de los paquetes - Inicio Conexión TCP

Véase a continuación el análisis de los paquetes de datos del procedimiento de establecimiento de conexión descrito en el apartado **Análisis de la máquina de estados TCP STATE**.

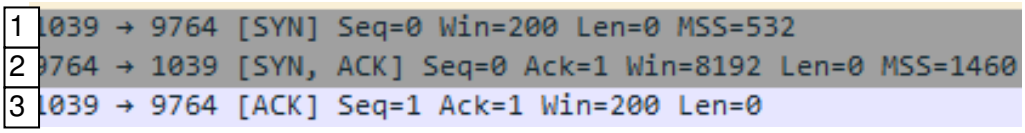


Fig. 7.21 Inicio conexión TCP. Fuente*: Conexion TCP.pcapng*

1) Inicio de conexión → [SYN] Seq=0 / Length = 0

En este caso, el MCU está iniciando la conexión por lo que el estado de la máquina de estados del stack corresponde con *TCP_SYN _SENT* de *TCP.c*

El microcontrolador envía un paquete de 60 bytes para inicializar el primer paso de la conexión (SYN). De estos 60 bytes, con las nociones de los distintos protocolos explicados en este proyecto, podemos reconocer los siguientes (Fig. 7.22)

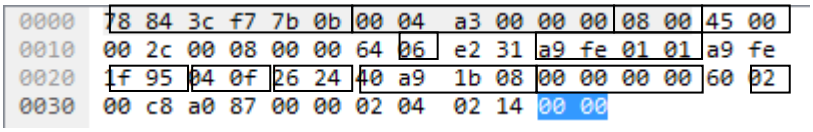


Fig. 7.22 Paquete 1 de Inicio de conexión abierto. Fuente: *Conexion TCP.pcapng*

78 84 3c f7 7b 0b (Bytes 0-5) → Dirección MAC de destino, el PC.

00 04 a3 00 00 00 (Bytes 6-11) → Dirección MAC de envío, el MCU.

08 00 (Bytes 12-13)→ Protocolo de Ethernet (IPv4).

06 (Byte 23)→ Protocolo TCP.

a9 fe 01 01 (Bytes 26-29→ Dirección IP de envío 169.254.1.1, el MCU.

a9 fe 1f 95 (Bytes 30-33)→Dirección IP de destino 169.254.31.149, el PC.

04 0f (Bytes 34-35)→Puerto de envío 1039. El stack asigna de forma aleatoria los puertos de envío. Por eso no son definidos en el código *GenericTCPClient.c*, ni en la propia configuración *TCPIP.h*.

26 24 (Bytes 36-37)→Puerto de destino 9764. Notar que, a diferencia del puerto de envío, el puerto de destino es definido dentro del código *GenericTCPClient.c* para que apunte al puerto abierto en el script de Python, que se ejecuta en el PC.

40 a9 1b 08 (Bytes 38-41)→SEQ. Es el número aleatorio de la secuencia relativa SEQ, necesaria para el control de la conexión. Por convención, y para que sea más sencillo seguir la secuencia de los distintos paquetes, el software Wireshark interpreta el número de secuencia inicial como 0, aunque el valor inicial transmitido es aleatorio y diferente a 0. *Seq=0.*

00 00 00 00 (Bytes 42-45) → ACK. Es el número *acknowdledgement* (acuse de recibo). Se inicializa a 0, a la espera de recibir el siguiente paso del establecimiento de la comunicación.

02 (Byte 49)→Flags de control. El segundo LSB del byte de control, corresponde al flag SYN.

Durante el establecimiento de la conexión el valor *Length* que controla la cantidad de bytes de información enviados es 0. Todavía no se ha empezado a trasmitir información. *Length=0*.

Una vez enviado el paquete, el MCU queda a la espera de la recepción del SYN / ACK.

2) Inicio de conexión → [SYN] / [ACK] Seq=0 / Ack=1 / Length = 0

EL PC ha recibido la petición de inicio de conexión. Responde enviando un paquete de 57 bytes, la estructura de los cuales es la misma que la analizada en la Fig. 7.19. En este paquete los bits de control SYN / ACK deben estar activos.

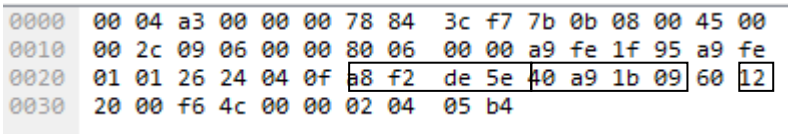


Fig. 7.23 Paquete 2 de Inicio de conexión abierto. Fuente: *Conexion TCP.pcapng*

Se analizan los flags de conexión y los bytes de Secuencia y acuse de recibo (acknowledgement).

a8 f2 d3 5e (Bytes 38-41)→SEQ. Es el número aleatorio de la secuencia relativa SEQ, necesaria para el control de la conexión. Al igual que sucede en el microcontrolador, el script inicializa el número de secuencia de manera aleatoria *Seq=0.*

40 a9 1b 09 (Bytes 42-45) → ACK. Durante el establecimiento de la conexión, el número de ACK corresponde al número SEQ recibido (*40 a9 1b 08)* más 1. Ack=1

12 (Byte 49)→Flags de control. El quinto bit y segundo LSB del byte de control, están activos, corresponden a los flags ACK y SYN respectivamente.

A modo aclaratorio, en caso de que el PC hubiese iniciado la conexión, el estado SYN /ACK del protocolo TCP queda reflejado en el MCU con el estado *TCP_SYN_RECEIVED* de la máquina de estados de *TCP.c*

3) **Inicio de conexión → [ACK] Seq=1 / Ack=1 / Length = 0**

El MCU ha recibido el SYN / ACK del PC y responde enviando el ACK con los numero Seq y Ack correspondiente.

En este caso, el número Seq debe corresponder con el número Ack recibido del PC y el núm Ack a enviar corresponderá con el número Seq recibido más uno.

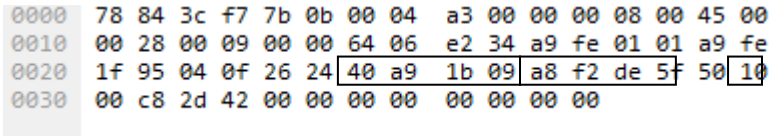


Fig. 7.24 Paquete 3 de Inicio de conexión abierto. Fuente: *Conexion TCP.pcapng*

40 a9 1b 09 (Bytes 38-41)→SEQ. Es el número aleatorio de la secuencia relativa SEQ, necesaria para el control de la conexión. Corresponde al número de secuencia enviado al iniciar la conexión más 1 *Seq=1.*

a8 f2 d3 5f (Bytes 42-45)→ ACK. Durante el establecimiento de la conexión, el número de ACK corresponde al número SEQ recibido (*a8 f2 d3 5e)* más 1. Ack=1

10 (Byte 49)→Flags de control. El quinto LSB del byte de control, correspondiente al flag ACK, está activo.

Análisis de los paquetes - Gestión de la conexión TCP

```
1039 → 9764 [PSH, ACK] Seq=1 Ack=1 Win=200 Len=17
9764 → 1039 [PSH, ACK] Seq=1 Ack=18 Win=16616 Len=17
1039 → 9764 [ACK] Seq=18 Ack=18 Win=183 Len=0
```

Tal y como se ha explicado en el apartado 7.4, la gestión de la conexión se hace a través de los números de secuencia (Seq), conocimiento (Ack) y del flag de recepción (ACK). Los números Seq y Ack son secuenciales a la cantidad de bytes transmitidos (LENGTH).

En el caso de esta prueba de conexión, el elemento a transmitir son los 17 bytes del mensaje:

TCPPutROMString (MySocket, (ROM BYTE*) " **Prueba de conex.**") ;

Estos son trasmitidos del puerto 1039 del MCU al puerto 9764 del PC. Al ser la primera transmisión de datos después del establecimiento de la conexión, los números Ack y Seq son interpretados como un 1, por conveniencia, en el software Wireshark.

Al ser recibidos por el PC, éste responde con un mensaje activando el flag ACK, en donde el número Ack se obtendrá de la suma de la longitud del mensaje recibido al número al número Ack (Ack=18). Además, se ha explicitado en el Script que éste envíe de vuelta los

datos recibidos. Esto hace que, además del flag ACK, se esté transmitiendo información en el paquete de datos (Len=17), por lo que será necesario que el MCU confirme, a su vez, la recepción de estos nuevos datos. (FIG)

El último paso es la confirmación de la recepción de los datos enviados del PC al MCU. El número de secuencia de este mensaje (Seq=18) ha aumentado en 17, ya que corresponde a número de bytes de información enviados, a su vez el número de Ack también ha aumentado en la misma cantidad, ya que se ha recibido el mismo mensaje que se ha enviado.

Nótese también la disminución del número Win=183, debido a la disminución en 17 bytes del espacio disponible en el buffer RX del socket.

Análisis de los paquetes - Finalización de la conexión TCP

Al igual que el establecimiento de la conexión, el cierre de una conexión TCP se lleva a cabo mediante una confirmación en 4 pasos en las que ambas partes de la conexión envían un mensaje con el flag de petición de cierre (FIN) y otro de confinación (ACK). A continuación se muestran todos los mensajes que forman parte de la finalización de la conexión.

```
9764 → 1039 [FIN, ACK] Seq=18 Ack=18 Win=16616 Len=0
```

```
1039 → 9764 [ACK] Seq=18 Ack=19 Win=183 Len=0
[TCP Window Update] 1039 → 9764 [ACK] Seq=18 Ack=19 Win=200 Len=0
1039 → 9764 [FIN, ACK] Seq=18 Ack=19 Win=200 Len=0
9764 → 1039 [ACK] Seq=19 Ack=19 Win=16616 Len=0
```

En este caso, una vez el script envía el mensaje de vuelta al MCU *" Prueba de conex.",* se procede a inicializar el cierre de la conexión, enviando un mensaje con el flag de finalización de la conexión (FIN). Este mensaje se ha enviado antes de la recepción de ACK perteneciente al mensaje de prueba, con lo que se deberá esperar a recibir el ACK del primer mensaje antes de poder finalizar la conexión.

Por otra parte el MCU, una vez confirma la recepción del mensaje *" Prueba de conex.",* envía la confirmación (ACK; Seq=18; Ack=19) para el cierre de la conexión, aumentando en 1 el número de Ack.

Como se trata de una confinación a 4 pasos, queda el envío de la petición de cierre (FIN / ACK; Seq=18; Ack=19) por parte del MCU, que finalmente es confirmada por el script (ACK; Seq=19; Ack=19), quedando la conexión cerrada y el script a la espera para el inicio de una nueva conexión.

Una vez finalizada la prueba entre el Cliente TCP (MCU) y el Servidor (PC), se realiza el mismo procedimiento con el script en Python haciendo este de cliente y el MCU de servidor.

Aunque hacer esta segunda prueba pueda parecer redundante, es necesario hacerla ya que la conexión final entre MCU1 y MCU2 no podrá ser monitorizada con un software para la adquisición de paquetes. Debido a ello es necesario asegurar el funcionamiento de cada MCU en su función del cliente o servidor por separado.

7.6. Implementación del código HTML

En esta tercera aplicación se procede a utilizar en el código una variable dinámica por sensor de temperatura (~temperatura1~; ~temperatura2~). El uso e implementación de las variables dinámicas ya ha sido mostrado en la aplicación MAQ1, por lo que no se explicará en las siguientes líneas.

Sí que es conveniente explicar cómo el usuario obtiene el control de las actualizaciones de las temperaturas mostradas en la web mediante un formulario con el método GET.

Para entender que es y cómo se utiliza el método GET, se deben explicar dos de los métodos principales (GET y POST) que existen en el protocolo HTTP para la petición de información entre el cliente y servidor web [23].

Método GET

- El cliente hace una petición de información de una fuente determinada. La página web de la MAQ1 constituye una petición de información y por lo tanto requiere de un método GET como se ha mostrado en la Fig. 5.8.
- Esta petición se envía mediante la URL→ http://169.254.1.1/temperatura.htm (Se requiere al servidor que muestre la página temperatura.htm)
- El envío de datos es visible, ya que se utiliza la URL
- La petición de datos puede parametrizarse, pero está limitada a los 2048 caracteres que admite la URL.

Método POST

- El cliente hace un envío de datos al servidor. Los datos son enviados dentro del cuerpo de un mensaje HTTP.
- Los datos enviados no son visibles.
- La parametrización de los datos se hace en el cuerpo del mensaje.
- No hay restricción de la longitud de datos a enviar.

Aunque el stack permite el envío de formularios POST no ha sido necesario su uso. Además, el uso de las funciones POST, supone un aumento considerable en el uso de la escasa memoria del microcontrolador.

A continuación, se diseña el código HTML de la web, con un formulario GET implementado en JavaScript.

```
▼<form method="get" action="temperatura.htm">
  ▼<div class="maq3">
      <b>Server1:</b>
    ▼<select name="Serv1">
        <option value="1">Actualiza</option>
        <option value="0">No Actualiza</option>
      </select>
      " 
      "
```

Fig. 7.25 Código del formulario GET. Fuente: *temperatura.htm*

Este formulario de la Fig. 7.25 envía una petición de actualización de temperatura mediante el uso de dos variables (*Serv1; Serv2*). Estas variables tomarán los valores (0;1), según sea seleccionada la opción del formulario *(Actualiza / No Actualiza*).

Implementación de un formulario GET en el Stack.

Una vez acabado el código HTML de nuestra página web se convierte en formato MPFS, tal y como se ha mostrado en la MAQ1. Seguidamente se proceden a implementar las funciones de las que hace uso nuestra página web en el servidor web (MCU1).

Al igual que en la MAQ1 la implementación se realiza en el archivo *CustomHTTP.c.* Debido a que la implementación de las variables dinámicas ya ha sido explicada en anteriores apartados, se procede a diseñar la máquina de estados que controla la petición de temperatura.(Fig. 7.26).

Vemos que la función *HTTPExecuteGET()* retornará HTTP_IO_DONE una vez se haya ejecutado correctamente , mientras haya necesidad de procesar datos de forma asíncrona la función deberá retornar HTTP_IO_WAITING.

Esta es una característica importante tanto del método GET como del POST (no utilizado en este proyecto), pues permite la ejecución de otras máquinas de estado de forma asíncrona a la máquina de estados HTTP. En este proyecto esto es utilizado para poner en marcha una segunda conexión TCP (*GenericTCPClient()*) con otro microcontrolador y obtener una segunda temperatura.

```
typedef enum
{
    HTTP_IO_DONE = 0u,
    HTTP_IO_NEED_DATA,
    HTTP_IO_WAITING
} HTTP_IO_RESULT;
```

Esto implica que la máquina de estados HTTP no pasará al siguiente estado y quedará en el estado SM_HTTP_PROCESGET hasta obtener la temperatura del MCU 2.

```
    SM_HTTP_IDLE = 0u,
    SM_HTTP_PARSE_REQUEST,
    SM_HTTP_PARSE_HEADERS,
    SM_HTTP_AUTHENTICATE,
    SM_HTTP_PROCESS_GET,
    SM_HTTP_PROCESS_POST,
    SM_HTTP_PROCESS_REQUEST,
    SM_HTTP_SERVE_HEADERS,
    SM_HTTP_SERVE_COOKIES,
    SM_HTTP_SERVE_BODY,
    SM_HTTP_SEND_FROM_CALLBACI
    SM_HTTP_DISCONNECT


    case SM_HTTP_PROCESS_GET:

        // Run the application callback HTTPExecuteGet()
        if(HTTPExecuteGet() == HTTP_IO_WAITING)
        {// If waiting for asynchronous process, return to main app
            break;
        }

        // Move on to POST data
        smHTTP = SM_HTTP_PROCESS_POST;
```

A continuación, se muestra la máquina de estados que debe seguir la implementación del método GET, a fin de actualizar las variables dinámicas donde se guardan las temperaturas a enviar al navegador.

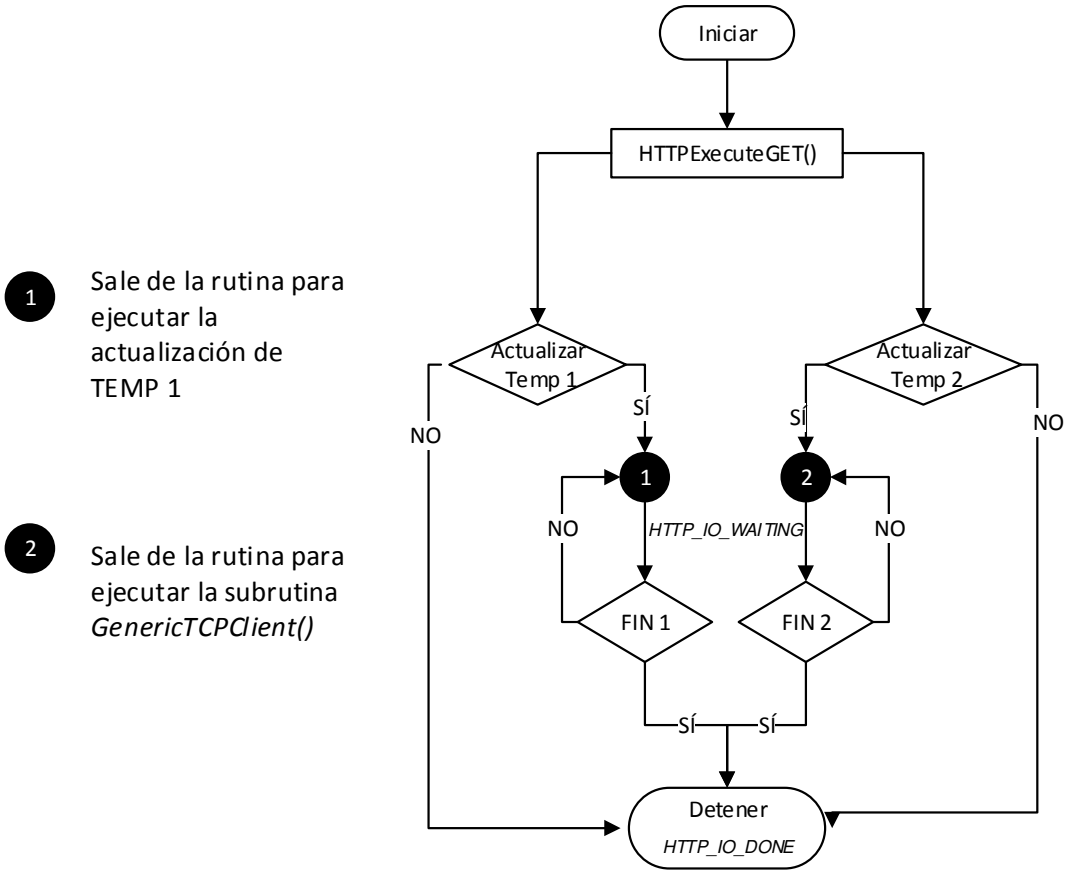


Fig. 7.26 Máquina de estados de la función GET para la actualización de temperaturas
Fuente: *CustomHTTP.c*

Para llevar a cabo el proceso indicado en la figura 7.25 deberemos leer la información enviada en la URL del navegador. Sin entrar en detalle de todas las funciones utilizadas, recalcar la utilización de la función *HTTPGetROMArg* la cual busca en la URL la existencia del argumento proporcionado y retorna un puntero a dicho argumento.

A partir de aquí solo es necesario comparar el valor del puntero que se acaba de guardar en RAM con el valor en ROM de dicho argumento (*stricmppgm2ram*).

En la figura 7.27 puede verse el código que ejecuta en *CustomHTTP.c* para implementar el metodo GET.

```
HTTP_IO_RESULT HTTPExecuteGet(void)
{
    BYTE *ptr;
    BYTE filename[20];
if((Actu2==0)&&(Actu1==0))
{
    MPFSGetFilename(curHTTP.file, filename, 20);
     if(!memcmppgm2ram(filename, "temperatura.htm", 15))
    {
        ptr = HTTPGetROMArg(curHTTP.data, (ROM BYTE *)"Serv1");
        if(ptr)
        {
            if(strcmppgm2ram((char*)ptr, (ROM char*)"1")==0)
            {
            Actu1 = 1;
            Final1=0;
            }
        }
        ptr = HTTPGetROMArg(curHTTP.data, (ROM BYTE *)"Serv2");
        if(ptr)
        {
            if(strcmppgm2ram((char*)ptr, (ROM char*)"1")==0)
            {
            Actu2 = 1;
            Final2=0;
            }
        }
    }
}
    if ((Final1==1)&&(Final2==1))
    {
        Actu2 = 0;
        Actu1 = 0;
        return HTTP_IO_DONE;
    }
    return HTTP_IO_WAITING;
}
```

Fig. 7.27 Función GET implementada en *CustomHTTP.c.* Fuente: *CustomHTTP.c*

7.7. Conexión y funcionamiento

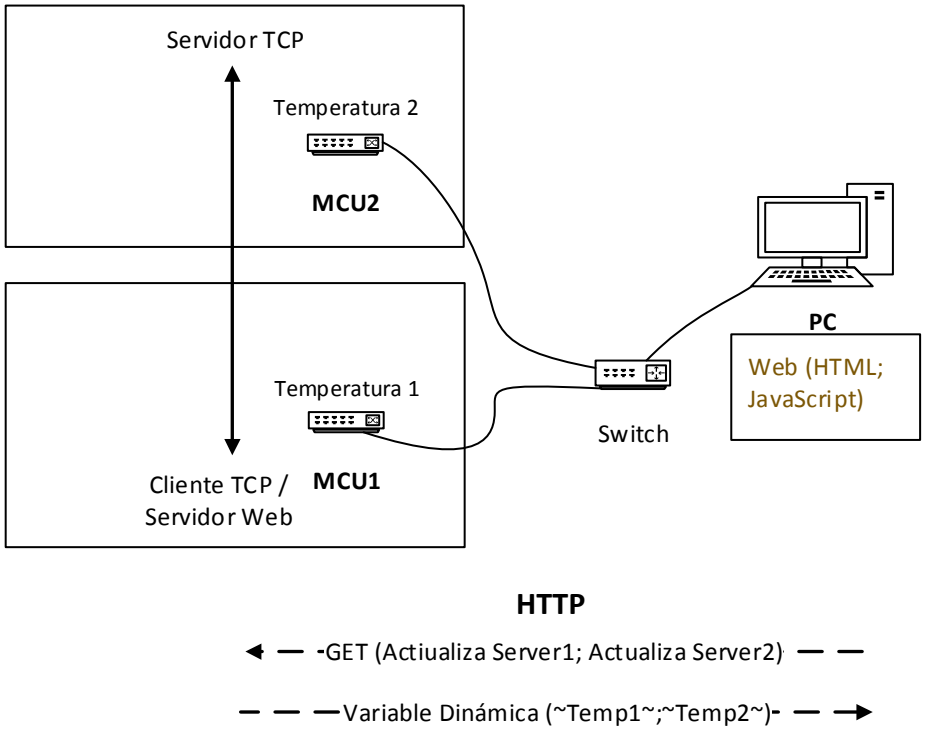


Fig. 7.28 Esquema de funcionamiento de la MAQ3. Fuente: Propia

Una vez conexionado tal y como se indica en el esquema se procede a ejecutar el script TCP Server.py y el proyecto MAQ3-Prueba Cliente TCP.mcw volcado en cada microcontrolador (MCU1

Probado el funcionamiento de los MCU por separado en su función de cliente y servidor, se procede al ensamblaje de la MAQ3 como tal.

En este caso, si bien la máquina de estados del cliente y del servidor codificada en los archivos *GenericTCPClient.c* y *GenericTCPServer.c* no cambiará, sí que lo hace el código implementado.

Recordemos que en esta última máquina queremos mostrar y controlar por un navegador web el acceso a la captura de temperaturas de dos sensores. En el anterior apartado se han explicado las modificaciones necesarias, tanto en el código del archivo *CustomHTTP.c*

como en el código HTML de la web, para implementar el uso de un formulario GET y la adición de una segunda variable dinámica.

Queda, por lo tanto, modificar el código en los archivos *GenericTCPClient.c* y *GenericTCPServer.c* para que transmitan la información que deseamos.

De esta manera, si arrancamos el servidor web y solicitamos que se nos muestre la web (http://169.254.1.1/) desde la dirección IP de nuestro PC (169.254.31.149). La web mostrada no contendrá, de inicio, ninguna temperatura, ya que para mostrar de forma didáctica el funcionamiento del método GET se ha desactivado la actualización automática de las variables de dinámicas que muestran la temperatura. (Fig. 7.29)

Source	Destination	Protocol	Length	Info
169.254.31.149	169.254.1.1	TCP	66	52090 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SA
169.254.1.1	169.254.31.149	TCP	60	80 → 52090 [SYN, ACK] Seq=0 Ack=1 Win=501 Len=0 MSS=532
169.254.31.149	169.254.1.1	TCP	54	52090 → 80 [ACK] Seq=1 Ack=1 Win=16616 Len=0
169.254.31.149	169.254.1.1	HTTP	441	GET / HTTP/1.1
169.254.1.1	169.254.31.149	TCP	60	80 → 52090 [ACK] Seq=1 Ack=388 Win=613 Len=0
169.254.1.1	169.254.31.149	TCP	60	[TCP Window Update] 80 → 52090 [ACK] Seq=1 Ack=388 Win=1
169.254.1.1	169.254.31.149	TCP	613	[TCP segment of a reassembled PDU]
169.254.1.1	169.254.31.149	TCP	219	[TCP segment of a reassembled PDU]
169.254.31.149	169.254.1.1	TCP	54	52090 → 80 [ACK] Seq=388 Ack=725 Win=16616 Len=0
169.254.1.1	169.254.31.149	HTTP	60	HTTP/1.1 200 OK (text/html)
169.254.31.149	169.254.1.1	TCP	54	52090 → 80 [ACK] Seq=388 Ack=726 Win=16616 Len=0
169.254.31.149	169.254.1.1	TCP	54	52090 → 80 [FIN, ACK] Seq=388 Ack=726 Win=16616 Len=0
169.254.1.1	169.254.31.149	TCP	60	80 → 52090 [ACK] Seq=726 Ack=389 Win=1 Len=0

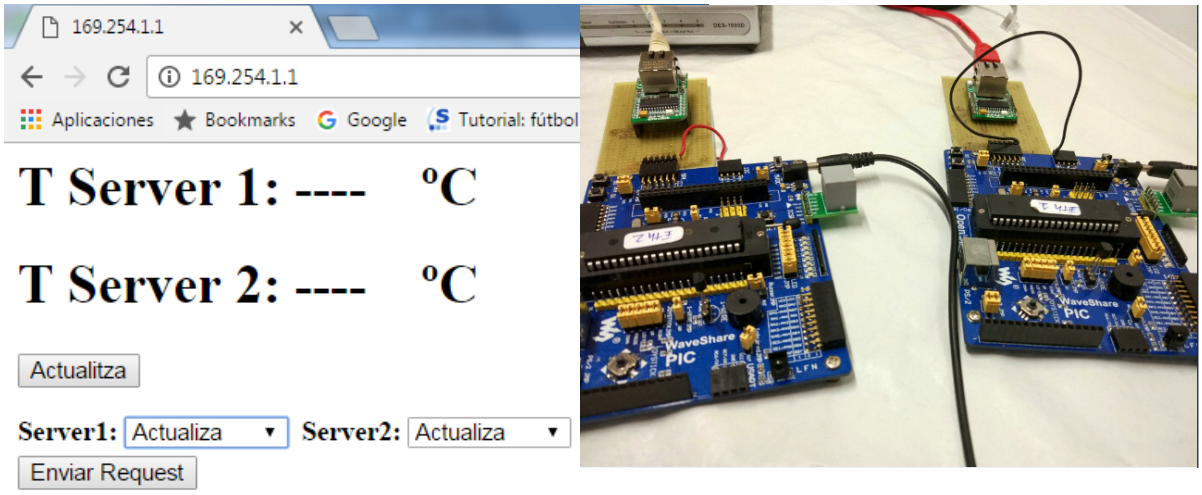


Fig. 7.29 Carga de web inicial. Fuente: MAQ3-Inicio.pcapng

Por el contrario, si enviamos una orden de actualización de temperaturas, las variables dinámicas se actualizan con el valor de la temperatura. (Fig. 7.30)

La temperatura 1 es recogida directamente del sensor del MCU1 mientras que la temperatura 2 debe de ser recogida del MCU2. Para ello se pone en marcha una subrutina servidor / cliente entre ambas placas. Recordar que, en este caso, no disponemos de los paquetes enviados entre los dos microcontroladores, de aquí radica la importancia del análisis de conexión llevado a cabo en el apartado 7.5.

169.254.31.149	169.254.1.1	TCP	66 52100 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PE
169.254.1.1	169.254.31.149	TCP	60 80 → 52100 [SYN, ACK] Seq=0 Ack=1 Win=501 Len=0 MSS=532
169.254.31.149	169.254.1.1	HTTP	502 GET /temperatura.htm?Serv1=1&Serv2=1 HTTP/1.1
169.254.1.1	169.254.31.149	TCP	60 80 → 52100 [ACK] Seq=1 Ack=449 Win=552 Len=0
169.254.1.1	169.254.31.149	TCP	60 [TCP Window Update] 80 → 52100 [ACK] Seq=1 Ack=449 Win=1000 L
169.254.1.1	169.254.31.149	TCP	60 [TCP Window Update] 80 → 52100 [ACK] Seq=1 Ack=449 Win=1 Len=
169.254.1.1	169.254.31.149	TCP	553 [TCP segment of a reassembled PDU]
169.254.1.1	169.254.31.149	TCP	283 [TCP segment of a reassembled PDU]
169.254.31.149	169.254.1.1	TCP	54 52100 → 80 [ACK] Seq=449 Ack=729 Win=16616 Len=0
169.254.1.1	169.254.31.149	HTTP	60 HTTP/1.1 200 OK (text/html)
169.254.31.149	169.254.1.1	TCP	54 52100 → 80 [ACK] Seq=449 Ack=730 Win=16616 Len=0
169.254.31.149	169.254.1.1	TCP	54 52100 → 80 [FIN, ACK] Seq=449 Ack=730 Win=16616 Len=0
169.254.1.1	169.254.31.149	TCP	60 80 → 52100 [ACK] Seq=730 Ack=450 Win=1 Len=0

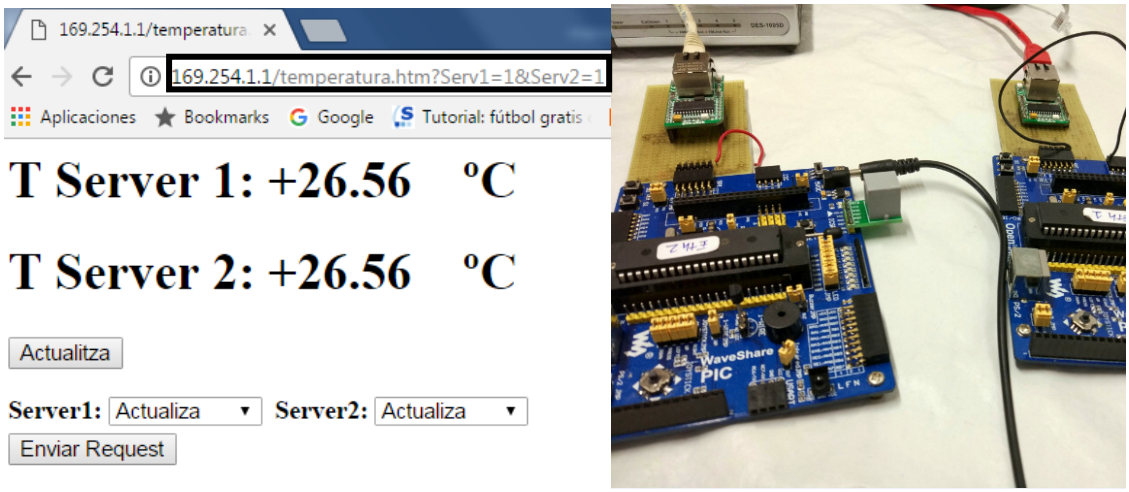


Fig. 7.30Fig. 7.29 Carga de web inicial. Fuente: MAQ3-Solicitar Temperaturas.pcapng

7.8. Uso de memoria y conclusiones.

Durante las pruebas de conexión realizadas entre el PC y el microcontrolador, solo se encontraba activo el protocolo TCP/IP, a pesar de ello, se puede observar en la figura 7.28 un uso de un 73% de la memoria de programa. Si lo comparamos con la figura 6.12, donde se muestra el uso de memoria del protocolo UDP (36%), vemos la gran diferencia de uso de memoria que comportan los dos protocolos.

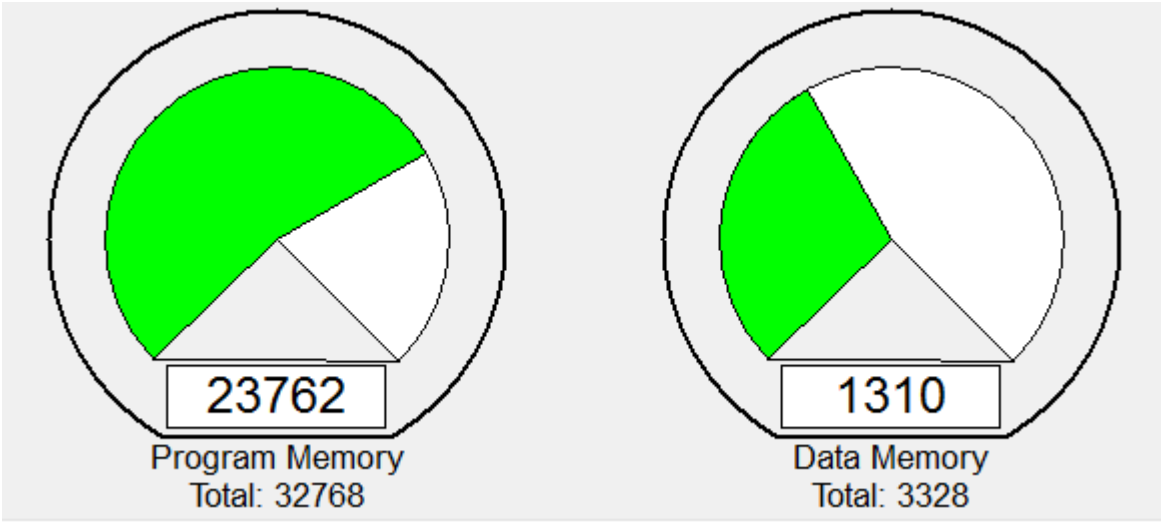


Fig. 7.31 Uso de memoria del MCU1 . Fuente: MAQ3-Prueba Cliente TCP.mcw

Por otra parte, la figura 7.29 muestra el uso en memoria de la MAQ3, una vez incluido el servidor web y la propia web es del 90%. Dejando de lado el peso en memoria de la página web, 745 bytes, podemos comprobar que el uso de la api *HTTP.c* queda limitado al método GET y a la inclusión de unas pocas variables dinámicas. Este microcontrolador no permite el uso de funciones más avanzadas.

Por lo tanto y como conclusión, el microcontrolador PICLF4680 puede utilizarse para implementar webs de control sobre variables específicas de una aplicación, mediante el metodo GET, siempre y cuando la memoria de programa que deje libre nuestra aplicación lo permita.

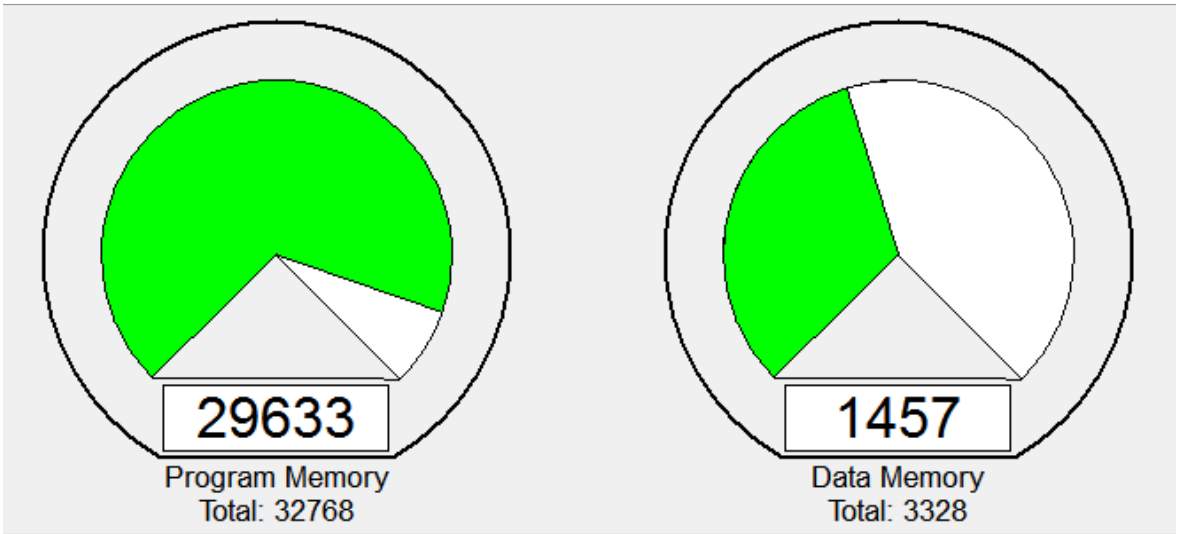
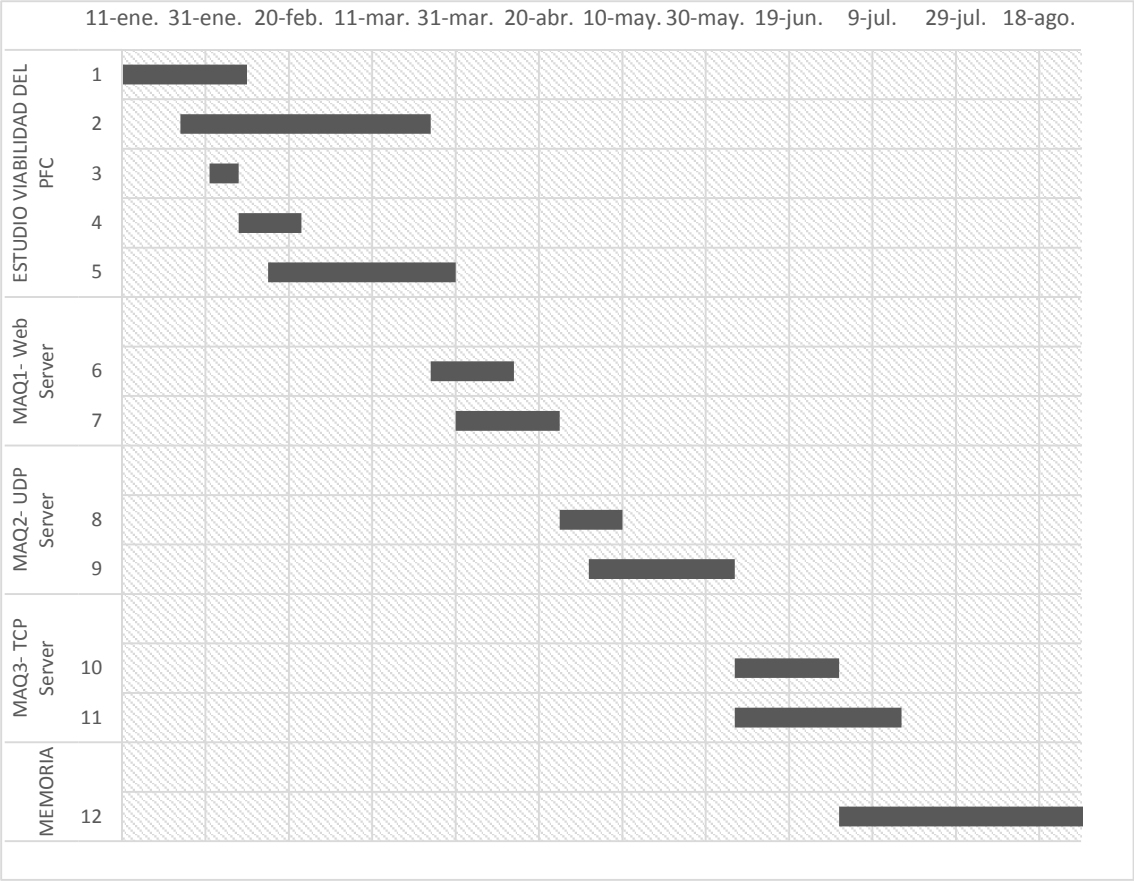


Fig. 7.30 Uso de memoria del MCU1 . Fuente: MAQ3-Web y Cliente TCP.mcw

8. Planificación

En este apartado se adjunta un diagrama de Gantt con la planificación inicial del proyecto, debido a los retos hardware a superar explicados en el apartado 3.3, las fases 4 y 5 del proyecto requirieron de más horas de las asignadas durante la planificación.



ESTUDIO VIABILIDAD DEL PFC	1	Estudio de la documentación del Stack (AN833)
	2	Estudio de la documentación de las MAL
	3	Diseño hardware
	4	Montaje de placa de hardware y validación
	5	Pruebas de compilación
MAQ1- Web Server	6	Diseño
	7	Montaje y prueba
MAQ2- UDP Server	8	Diseño
	9	Montaje y prueba
MAQ3- TCP Server	10	Diseño
	11	Montaje y prueba
MEMORIA	12	Redacción Memoria

9. Análisis Económico.

En este apartado se pasa a realizar una pequeña valoración de los costes asociados al proyecto.

Esta valoración se divide en dos apartados, el coste de los materiales utilizados para el desarrollo del proyecto y el coste de desarrollo del proyecto en sí mismo.

9.1. Coste de los materiales.

En este apartado se considerará el coste total del hardware y los costes de amortización del equipo. No se han tenido en cuenta los costes del software, ya que se trata en su totalidad software gratuito. No se han tenido en cuenta los costes de amortización del equipo secundario, como soldador, tester, etc.

Costes del hardware.

CONCEPTO	UND	COSTE [€]
Microcontrolador PIC18LF4680 de Microchip	2	14,60
Microcontrolador PIC18F4680 de Microchip	2	14,50
Controlador ENC28J60 Ethernet Click de MikroElektronika	2	43,40
Sensor DS18B20 de Maxim Integrated	2	16,30
Programador ICD 3 Microchip	1	178,50
Pack Open18F4520 Package B	2	147,50
Placa de topos, conectores macho, conectores hembra y cables	varios	16,00
	SUBTOTAL	**430,90 €**

Costes de amortización del equipo.

CONCEPTO	COSTE[€]	AMORTIZACIÓN	COSTE TOTAL[€]
Ordenador Personal	1100,00	30%	330,00
		SUBTOTAL	**330,00 €**

Consumo

CONCEPTO	HORAs	COSTE [€/KWh]	COSTE TOTAL[€]
Ordenador Personal (90W)	400,00	0,12	36,00
		SUBTOTAL	**36,00 €**

9.2. Coste de desarrollo.

A los costes materiales deben añadirse los costes del tiempo dedicado al diseño, montaje y desarrollo del hardware y software de las placas, así como los de validación de las aplicaciones.

CONCEPTO	COSTE HORA [€]	COSTE TOTAL [€]
Diseño\ Montaje \ Validación del hardware (150 h)	40,00	6000,00
Estudio\Diseño\Programación(280 h)	40,00	11200,00
Redacción (180 h)	40,00	7200,00
	SUBTOTAL	**24400,00€**

9.3. Coste total del proyecto

El coste total del proyecto

CONCEPTO	COSTE TOTAL [€]
Costes del hardware	430,90
Costes de amortización	330,00
Costes de consumo	36,00
Costes de desarrollo	24400,00
SUBTOTAL	**25196,90€**

10. Estudio del Impacto ambiental

Debido a que este proyecto se centra en una visión claramente didáctica para con el uso y desarrollo de aplicaciones que utilicen protocolos de comunicación Ethernet, el estudio de impacto ambiental debe desarrollarse en base a la pregunta de qué han aportado/cambiado estos protocolos según la Directiva 2011/92/UE

Es por ello que el estudio del impacto ambiental realizado para este proyecto consiste en una evaluación de los cambios que el uso de estos protocolos de comunicación conlleva en la industria y más concretamente en el montaje de vehículos a motor, previsto en el anexo II de esta misma directiva.

Desde el desarrollo del bus CAN (Controlled Area Network) en 1986 la tendencia dentro de la industria automovilística ha sido la de utilizar, cada vez más, la intercomunicación de las distintas partes del vehículo. Tanto para la gestión de elementos motores (ECU), de seguridad o multimedia. Lo que ha conllevado un mayor uso de elementos materiales y recursos energéticos para la construcción y montaje de los vehículos.

Ya desde hace unos años y debido a los cada vez mayores requerimientos de intercomunicación, se han venido desarrollando distintos protocolos (LIN, FlexRay, MOST) encargados de distintos niveles de intercomunicación.

En la actualidad, la tendencia es el desarrollo de tecnologías basadas en Ethernet dentro de la industria automovilística, como el TTEthernet (Time Triggered Ethernet) [24], que junto con la substitución de los cables LVDS (low-voltage differential signaling) por cables de Ethernet podría reducir los coste de conectividad en un 80% y el peso del cableado en un 30% según Broadcom [25].

Así pues, el uso de protocolos basados en Ethernet puede tener impacto positivo en la reducción de los costes materiales y energéticos de la industria automovilística.

Conclusiones

El objetivo del presente proyecto ha sido realizar un estudio de las limitaciones de un microcontrolador de 8 bits para su uso conjunto con un stack de comunicaciones TCP/IP.

A lo largo del proyecto se ha mostrado la dificultad de implementar funciones avanzadas de comunicación debido a los requerimientos de memoria y de potencia de cálculo necesarios para llevarlos a cabo.

No obstante, y teniendo en cuenta las limitaciones del microcontrolador, se han diseñado e implementado tres aplicaciones que han servido para explicar, de una manera didáctica, no solo las características principales de los protocolos del modelo de comunicaciones TCP/IP, sino también para dar forma a tres demostraciones, que pueden servir como base para futuras prácticas del alumnado del Grado o del Máster Universitario de Ingeniería Industrial.

Con la idea de servir como base didáctica, todo el hardware utilizado puede encontrarse en los laboratorios del Departamento de Ingeniería Electrónica de la UPC (sección sud / ETSEIB), con la salvedad del PIC18 utilizado, que se corresponde con la versión de bajo voltaje y alta capacidad PIC18LF4680. El software utilizado es de libre distribución, haciendo posible la fácil repetitividad de los resultados mostrados.

Las funcionalidades mostradas a lo largo del proyecto han sido:

- El uso del stack para monitorizar variables en tiempo real a través de una página web mediante el diseño de aplicaciones HTTP.
- El análisis del protocolo UDP, de reducidos requerimientos en la memoria del PIC y su adecuación para determinadas comunicaciones entre dispositivos.
- El uso del software Python para la creación de scripts con distintos sockets de comunicación.
- El protocolo TCP y el modelo de la gestión de la conexión. El análisis de su funcionamiento y su uso en el PIC.
- El uso del método GET para la recolección de información de forma asíncrona entre dispositivos.

Todo ello se ha descrito mediante máquinas de estado con las que poder repetir y ampliar las funcionalidades mostradas.

Bibliografía

Referencias bibliográficas

[1] ISO ORG. *Information technology -- Open Systems Interconnection -- Basic Reference Model: The Basic Model.* [http://www.iso.org/iso/catalogue_detail.htm?csnumber=20269 , Febero 2016]

[2] GUILLERMO RIGOTTI. FACULTAD DE CIENCIAS EXACTAS (UNICEN). *Apuntes Comunicación de Datos I* p. 7-9.* [www.exa.unicen.edu.ar/catedras/comdat1/material/ElmodeloOSI.pdf, Enero 2016]

[3] NILESH RAJBHARTI. MICROCHIP TECHNOLOGY INC. *The Microchip TCP/IP Stack – AN833. p. 1-2.*

[4] MICROCHIP TECHNOLOGY INC. *PIC18F2585/2680/4585/4680 Data Sheet.*

[5] MAXIM INTEGRATED. *DS18B20 Datasheet. Programmable Resolution 1-Wire Digital Thermometer.*

[6] MICROCHIP TECHNOLOGY INC. *ENC28J6 Datasheet. Stand-Alone Ethernet Controller with SPI™ Interface.*

[7] Open18F4520 Standard PIC Development Board [h*ttp://www.waveshare.com/open18f4520-standard.htm, Agosto 2016*]

[8] MICROCHIP TECHNOLOGY INC. *PIC18F2585/2680/4585/4680 Data Sheet.* p. 4.

[9] MICROCHIP TECHNOLOGY INC. *ENC424J600/624J600 Datasheet Stand-Alone 10/100 Ethernet Controller with SPI™ Interface or Parallel Interface.* p. 3.

[10] MIKROELEKTRONIKA- *ETH Click Manual.* [*http://www.mikroe.com/downloads/get/1746/eth-click-manual-v100.pdf; Agosto 2016]*

[11] ETH Click [http://www.mikroe.com/click/eth/ , Febrero 2016]

[12] MICROCHIP TECHNOLOGY INC. *PIC18F2585/2680/4585/4680 Data Sheet.* p. 415-416.

[13] *MANUEL MORENO.* UPC BARCELONA TECH. *DS18B20 Lib User Manual - C Library for the DS18B20 Digital Thermometer.*

[14] JON POSTEL – *User Datagram Protocol* [https://www.rfc-es.org/rfc/rfc0768-es.txy , Abril 2016].

[15] *MICROSOFT Tamaño de mensaje UDP máximo diferente de lo esperado en Windows 2000 SP3 [https://support.microsoft.com/es-es/kb/822061, Abril 2016]*

[16] INFORMATION SCIENCE INSTITUTE. UNIVERSITY OF SOUTHERN CALIFORNIA – *Protocolo de Control de Transmisión* [https://www.rfc-es.org/rfc/rfc0793-es.txt, Junio 2016].

[17] M. SIMMONS-MICROCHIP TECHNOLOGY INC. *Ethernet Theory of Operation– AN1120. p. 8-9. [http://www.microchip.com/wwwAppNotes/AppNotes.aspx?appnote=en533903, Enero 2016]*

[18] *DAVID C. PLUMMER- An Ethernet Address Resolution Protocol.* [https://tools.ietf.org/html/rfc826, Mayo 2016]*

[19] *GODRED FAIRHURST- ELECTRONICS RESEARCH GROUP- UNIVERSITY OF ABERDEEN. Address Resolution Protocol (arp) [http://www.erg.abdn.ac.uk/users/gorry/course/inet-pages/arp.html, Junio 2016]*

[20] INFORMATION SCIENCES INSTITUTE - UNIVERSITY OF SOUTHERN CALIFORNIA– *Protocolo de Control de Transmisión p. 20-21* [https://www.rfc-es.org/rfc/rfc0793-es.txt, Mayo 2016] .

[21] INFORMATION SCIENCES INSTITUTE - UNIVERSITY OF SOUTHERN CALIFORNIA– *Protocolo de Control de Transmisión p. 22-32* [https://www.rfc-es.org/rfc/rfc0793-es.txt , Mayo 2016] .

[22] THEO SCHOUTEN - INSTITUTE FOR COMPUTING AND INFORMATION SCIENCES - 6.5.5 - 6.5.7 TCP connection management* [http://www.cs.ru.nl/~ths/a3/html/h6/h6.html, Junio 216]

[23] HTTP Methods: GET vs. POST [http://www.w3schools.com/tags/ref_httpmethods.asp , Mayo 2016]

[24] LUCIA LO BELLO – *The case for ethernet in automotive communications.* ACM SIGBED Review - Special Issue on the 10th International Workshop on Real-time Networks (RTN 2011)

[25] BROADCOM *The Case for Ethernet in Cars* [http://www.broadcom.com/blog/automotive-technology-2/the-case-for-ethernet-in-cars , Agosto 2016]