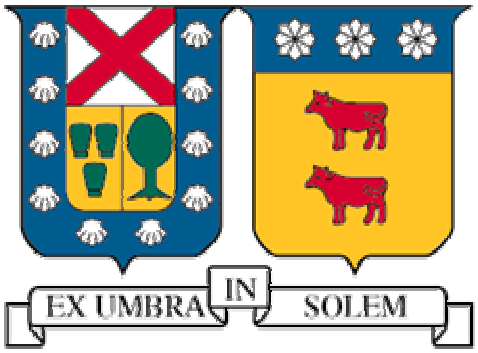
Universidad Técnica Federico Santa María
Departamento de Electrónica



"Desarrollo de aplicaciones para la conectividad IP de microcontroladores"

Memoria presentada por: Pablo Hernán Naveas Farías
como requisito parcial para optar al título de
Ingeniero Civil Electrónico
Mención Computadores

Profesor Guía: Ing. Leopoldo Silva Bijit

Valparaíso, Noviembre 2004

Dedicado a la memoria de mi querida Yaya y de mi abuelo Carlos.

Resumen

En la actualidad, existen diversas alternativas para efectuar la conexión de un sistema embebido (dispositivo gobernado por un microprocesador) a una red de computadores y a Internet. Cualquiera que sea el método escogido, la posibilidad de realizar el control de un proceso y la adquisición de datos en forma remota, a través de redes IP, es hoy en día una opción muy atractiva. En la introducción de este trabajo se entrega una reseña general acerca de la denominada *Embedded Internet*, sus proyecciones y la breve descripción de algunas alternativas de implementación.

En los siguientes cuatro capítulos se detalla el proceso de desarrollo e implementación de aplicaciones para la conectividad IP de un sistema en particular.

El capítulo 1 entrega la descripción de la interfaz de red del hardware utilizado: la tarjeta de desarrollo Easyweb, con un microcontrolador de la familia MSP430, de Texas Instruments. Se describe a su vez el stack de protocolos TCP/IP uIP, escogido como base para el desarrollo de aplicaciones de red, cuyo método de programación se encuentra en los capítulos 2 y 3. Se trata de programas en lenguaje C que corren sobre el stack TCP/IP y permiten la transferencia y el procesamiento de datos a través de conexiones TCP.

En forma paralela, se realizó el desarrollo de aplicaciones de usuario. Estos son programas en lenguaje Java, para su ejecución desde un PC, que permiten establecer una conexión, controlar y adquirir datos desde la tarjeta, en forma remota. Finalmente, con el objetivo de efectuar lo anterior a través de un browser, se propone el uso de servlets y JSP, con un servidor web como intermediario e involucrado en el establecimiento y administración de las conexiones con la tarjeta. El detalle de estas aplicaciones web y la arquitectura propuesta se encuentran en el capítulo 4.

Índice

Introducción. 1
I. Conectividad en red de microprocesadores.....1
II. El Protocolo TCP/IP.....2
III. Modos de incorporar TCP/IP a un sistema embebido.....3
IV. Implementaciones de TCP/IP para sistemas embebidos.....4
Capítulo 1: Tarjeta de desarrollo Easyweb y su Interfaz de Red.....6
1.1. Introducción.....6
1.2. Interfaz de Red de la Tarjeta Easyweb2.....7
1.2.1. Conexión del chip CS8900A al MSP430F149..... 7
1.2.2. Conexión CS8900 a componentes externos.....8
1.2.3. Transmisión y recepción de paquetes con el CS8900A.....9
1.2.4. Packet Driver para el CS8900A..... 9
1.2.5. Stack TCP/IP..... 12
Capítulo 2: Uso de la API del stack uIP.....13
2.1. API (*Application Program Interface*).....13
2.2. Desarrollo de la primera aplicación.....14
2.2.1. Recepción y envío de un paquete IP.....16
2.2.2. Recepción y envío de un paquete ARP.....17
2.2.3. Procesamiento periódico.....17
2.2.4. Inicialización.....18
2.3. Conexión a la red y configuración de cliente TCP/IP.....20
2.4. Conexión con un cliente TCP/IP.....21
2.4.1. Cliente TCP/IP en Windows.....21
2.4.2. Programación de un Cliente TCP/IP.....22
Capítulo 3: Diseño estructurado de aplicaciones.....25
3.1. Administrador de eventos.....25
3.2. Programación de una aplicación estructurada.....26
3.3. Cliente SMTP para envío de e-mail.....31
3.3.1. Establecimiento de la conexión y preparación del mensaje.....31
3.3.2. Interfaz de la aplicación.....36
3.3.3. Término del envío y cierre de la conexión.....38
3.3.4. Análisis del tráfico de paquetes.....39
Capítulo 4: Control en forma remota de la tarjeta Easyweb2.....41
4.1. Sockets y streams.....41
4.2. Programación de un Cliente TCP/IP con interfaz gráfica de usuario.....42
4.2.1. Creación del JFrame y de los componentes gráficos.....42
4.2.2. Programación de los eventos para las componentes de la GUI.....47
4.2.3. Programación del servidor.....50
4.3. Desarrollo de aplicaciones para su ejecución en un servidor web.....55
4.3.1. Servlets.....56
4.3.2. Programación de una aplicación web: HTML y servlet.....58
4.3.2.1. Diseño y despliegue de la página web.....59
4.3.2.2. Programación del servlet.....60
4.3.2.3. Programación de la aplicación servidor Easyweb2.....65
4.3.2.4. Ejecución de la aplicación web.....67
4.3.3. Aplicaciones web mediante plantillas (*templates)*.....68
4.3.3.1. Páginas JSP [30].....68
4.3.3.2. Componentes JavaBeans.....69

4.3.3.3. JavaServer Faces [32].....69
4.3.3.4. Selección de IDE y programación de aplicación web final.....70
4.3.3.5. Páginas JSP y reglas de navegación.....71
4.3.3.6. Estructura de directorios del proyecto.....72
4.3.3.7. Preparación de las páginas y programación de los Beans.....73
Conclusiones y trabajos futuros.....79
Referencias.....81
Anexo.....83

Índice de Figuras

Introducción.

Figura I: Número de computadores vendidos entre 1998-2002....1

Capítulo 1.

Figura 1.1: Tarjeta de desarrollo Easyweb2....6
Figura 1.2: Esquemático conexión MSPF149 a CS8900A en tarjeta Easyweb2....7
Figura 1.3: Esquemático conexión CS8900A a componentes externos....8

Capítulo 2.

Figura 2.1: Archivos uIP para desarrollo de una aplicación de red....15
Figura 2.2: Diagrama de loop para control de la aplicación....16
Figura 2.3: Conexión de tarjeta Easyweb a la red.... 20
Figura 2.4: Prueba de conectividad mediante ICMP.... 20
Figura 2.5: Cliente TCP/IP Winsock.... 21
Figura 2.6: Tráfico de paquetes en establecimiento de conexión TCP/IP....21
Figura 2.7: Conexión TCP/IP establecida e intercambio de datos....22
Figura 2.8: Streams y socket de cliente TCP/IP....23
Figura 2.9: Ejecución del cliente "TCPClient"....24

Capítulo 3.

Figura 3.1: Diagrama de estados de la aplicación....26
Figura 3.2: Esquema de estructura `app2_struct`....27
Figura 3.3: Conexión con cliente TCP/IP....30
Figura 3.4: Diagrama del cliente SMTP para envio de correo electrónico....32
Figura 3.5: Despliegue inicial, cliente SMTP....36
Figura 3.6: Conexión establecida con el servidor....36
Figura 3.7: Envío exitoso del e-mail....38
Figura 3.8: Tráfico de paquetes aplicación cliente SMTP....39

Capítulo 4.

Figura 4.1: Esquema de un socket y flujo de datos a través de la conexión de red....41
Figura 4.2: Creación del JFrame....43
Figura 4.3: Ambiente de desarrollo de la GUI....43
Figura 4.4: Código generado por el IDE para la clase TCPClient....44
Figura 4.5: Insertar un componente en el Jframe de la GUI....45
Figura 4.6: Código generado para el componente jButton....46
Figura 4.7: Componentes de la GUI.... 46
Figura 4.8: Programación de eventos para un componente de la GUI....47
Figura 4.9: Código de atención de eventos por componente de la GUI....49
Figura 4.10: Aplicaciones cliente GUI – servidor Easyweb....54
Figura 4.11: Esquema de la aplicación web a implementar....57
Figura 4.12: Iniciando el servidor web incorporado a NetBeans IDE 3.6....58
Figura 4.13: Creación de la estructura de directorios de la aplicación web....58
Figura 4.14: Ubicación de los archivos en la estructura de directorios del módulo web....59
Figura 4.15: Página web para el ingreso de los datos de conexión....60
Figura 4.16: Estructura de directorios de la aplicación web y métodos por omisión del servlet....61
Figura 4.17: Código de la clase recién creada easyServlet.java....62
Figura 4.18: Página de respuesta enviada por el servlet....67
Figura 4.19: Procesamiento de una página JSP en el servidor....68
Figura 4.20: Visibilidad (*scope*) de un Bean dentro de la aplicación web....69
Figura 4.21: Esquema del IDE para las reglas de navegación....71
Figura 4.22: Ventana de navegación del proyecto....72
Figura 4.23: Página de bienvenida....73
Figura 4.24: Panel de control....76
Figura 4.25: Ejecutar aplicación web desde Studio Creator....78

Introducción.

I. Conectividad en red de microprocesadores. *(Embedded Internet)*

Desde hace dos décadas, los microprocesadores se han ido incorporando progresivamente en la vida cotidiana de las personas y en sus principales actividades productivas. Juguetes, electrodomésticos, teléfonos celulares, agendas electrónicas, equipos médicos, son sólo algunos ejemplos de lo primero. Con respecto a lo segundo, el poder contar con estos dispositivos electrónicos en la actividad industrial se ha convertido en algo muchas veces imprescindible, o por lo menos en una garantía de mejor eficiencia, seguridad y dinamismo de los diversos procesos productivos del hombre.

La conectividad de estos dispositivos es necesaria ya sea para la obtención y transferencia de información, o para acceder al control y la configuración de los dispositivos en forma remota.

Hasta la segunda mitad de 1996, la interconexión de estos dispositivos se realizaba principalmente a través de puertas seriales vía UART. La posibilidad de acceder a ellos en forma remota estaba limitada a transmisores de RF de este protocolo, o bien a la incorporación de módems. Pero fue a fines de ese año que surge la idea de conectar estos dispositivos entre sí a través de redes de paquetes IP. Sistemas embebidos conectados entre sí y con otros computadores a través de Internet, gracias a la implementación de un conjunto de protocolos denominado TCP/IP.

Un sistema embebido se define como un computador (cuyo núcleo es un microprocesador) instalado adentro de otro dispositivo, como una lavadora, un automóvil o un televisor. Es decir, los sistemas embebidos forman en su totalidad la mayor clase de computadores, y abarcan la más amplia gama de aplicaciones y desempeño. Por esto, y desde hace varios años, el número existente de computadores embebidos supera por mucho al de PCs de escritorio y servidores (fig. I.) [1]:

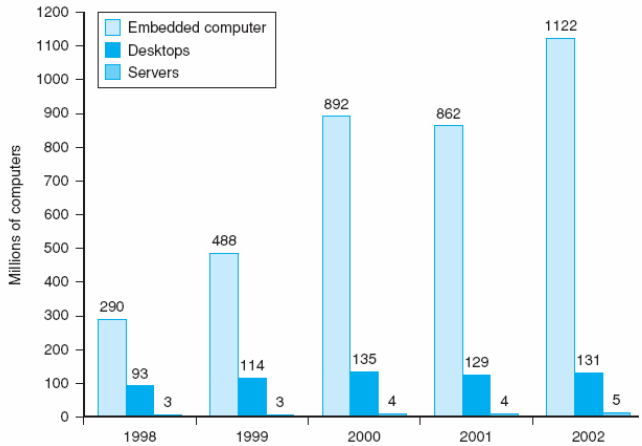


Fig. I. Número de computadores vendidos entre 1998-2002.

Cabe señalar que en el gráfico anterior no se incluye a procesadores de 8 y 16 bits, pero igual da cuenta del impacto que, sólo por cantidad, tiene la incorporación de sistemas embebidos a la red.

Hoy en día existe un consenso general respecto a que, en los años venideros, los dispositivos conectados a Internet estarán embebidos más que orientados al escritorio. Incluso el slogan "En el 2010, el 95% de los sistemas conectados a Internet serán embebidos, no PCs" está siendo utilizado por empresas que ofrecen sus soluciones de conectividad para sistemas embebidos [2].

Así como muchos usuarios olvidan la presencia de microprocesadores en los artefactos de uso cotidiano, el hecho que éstos además se conecten a Internet puede en muchos casos también llegar a pasar desapercibido. Esto otorga otra dimensión al empleo de esta tecnología: quiebra la brecha de necesitar un PC para ingresar a la red y acerca la Internet de un modo 'invisible' a la vida cotidiana de la gente.

II. El protocolo TCP/IP

TCP/IP (Transmission Control Protocol/Internet Protocol) es el estándar para las comunicaciones a través de redes de computadores. Corresponde a un grupo o stack de sub-protocolos, que permiten llevar a cabo el transporte de los datos que se desean transferir a través de la Internet. Algunos ejemplos de estos sub-protocolos, agrupados por capas según el modelo OSI de redes son:

- Capa de enlace de datos: ARP (*Address Resolution Protocol*)
- Capa de red: ICMP (*Internet Control Message Protocol*), IP.
- Capa de transporte: TCP, UDP (*User Datagram Protocol*).
- Capa de aplicación: TELNET (*Terminal Emulation Protocol*), HTTP (*Hypertext Transfer Protocol)*, SMTP (*Simple Mail Transfer Protocol*).

No siendo el objetivo profundizar acerca de estos u otros protocolos en particular, basta con señalar que la implementación de este stack (o al menos de una parte de él), sobre una plataforma con menores recursos de hardware que un PC, como es un microprocesador, es factible hoy en día. Sin embargo, es conveniente revisar el contexto histórico que rodea al stack TCP/IP.

Los protocolos TCP/IP fueron desarrollados por DARPA (*Defense Advanced Research Projects Agency*) de Estados Unidos, para conectar mediante una red de computadores las principales instituciones de investigación del país que recibieran financiamiento gubernamental. Paralelamente, en la Universidad de Berkeley surge BSD (Berkeley Systems Distributions), una distribución de UNIX que es rápidamente adoptada por los restantes departamentos de ciencia de computadores del país. Para mantener un conjunto estándar de protocolos en lo concerniente a la conectividad en red, DARPA propone que BSD incluya el stack TCP/IP como parte del sistema operativo. Fruto de esto, el conjunto de protocolos que conforman el stack TCP/IP son distribuidos en la forma de los códigos fuente de BSD con los derechos de autor de la Universidad de Berkeley. A partir de este código fuente, diferentes empresas de software han portado estos protocolos para

que cumplan con los requerimientos de un sistema embebido, adaptando y modificando el código original.

Algunos de estos requerimientos, que caracterizan a un sistema embebido de tiempo real, son los siguientes [3]:

- Memoria limitada: por esta razón, una correcta administración de los buffers de almacenamiento de los paquetes debe evitar el uso dinámico de memoria, reservando un espacio fijo en RAM para este efecto.

- Uso de Timers: esenciales para la administración de la conexión (tiempo de vida de paquetes, retransmisiones). Si no son manejados por un sistema operativo de tiempo real, debe minimizarse el uso de CPU en el manejo de timers para evitar problemas de latencia.

- Minimizar la copia de datos: si cada trama se almacena en un único buffer, se evita que la CPU deba ocuparse en copiar los datos a cada etapa del protocolo.

III. Modos de Incorporar TCP/IP a un sistema embebido.

Dependiendo de los requerimientos de conectividad del diseño que se quiere realizar, surgen distintas soluciones a la hora de implementar un stack TCP/IP. Por ejemplo, puede que se necesite un dispositivo que se utilice para la captura y posterior análisis de datos. Puede pretenderse también acceder al dispositivo a través de un browser, para lo cual debe tener montado un servidor web.

En la práctica, las maneras de incorporar TCP/IP a un proyecto son las siguientes [3]:

- Implementación por hardware: el sistema embebido logra la conectividad a redes IP gracias a la conexión vía puerto serial a otro dispositivo, que es el que en definitiva se conecta a la red. Ejemplos de esto son Connect One o XPort[4].

- Programar un stack propio: tarea compleja, y en la mayor parte de los casos específica para un diseño en particular.

- Escoger un stack TCP/IP y un sistema operativo en tiempo real por separado y eventualmente trabajar en la integración de ambos productos.

- Adquirir la solución completa: sistema operativo con stack TCP/IP incorporado.

IV. Implementaciones de TCP/IP para sistemas embebidos (*Embedded Internet Systems*).

La siguiente lista da una idea de la oferta que existe actualmente al momento de adquirir un stack TCP/IP para el desarrollo de sistemas embebidos con conectividad a Internet:

- KwikNet TCP/IP stack [5]: disponible para varias familias de microprocesadores. Ofrece también los drivers para los controladores Ethernet más comunes.

- SMX RTOS [6]: sistema operativo de tiempo real, con stack TCP/IP incorporado y con soporte para varias familias de microprocesadores.

- Fusion TCP/IP [7]: ofrecido como una solución altamente portable, es independiente tanto del compilador como del sistema operativo utilizado y está programado en ANSI C.

- Interpeak IPNet [8]: completo stack TCP/IP para sistemas embebidos que garantiza la compatibilidad con IPv6.

- NetBurner TCP/IP [9]: ofrece su solución de red en el marco de un completo kit de desarrollo, que incluye un sistema operativo de tiempo real, su propio compilador y depurador, más aplicaciones de red ya programadas. Todo esto con el objetivo de reducir el tiempo de desarrollo de aplicaciones propias.

- TargetTCP [10]: con características similares a los anteriores y también programado en C, garantiza la interoperabilidad con otros stack TCP/IP.

- CMX Micronet [11]: la principal diferencia con los anteriores es que es el único de la lista y uno de los pocos que tiene soporte específico para los microcontroladores MSP430 de Texas Instruments, que es el hardware con que se cuenta para este trabajo.

Si lo que se busca es código abierto, algunas de las opciones son:

- uCLinux [12] : símbolo del open source, Linux es portado a los microcontroladores con todas sus funciones de red, en distribuciones como uCLinux.

- OpenTCP [13]: proyecto de desarrollo de un stack TCP/IP para microcontroladores de 8 y 16 bits.

- uIP [14]: portado a diferentes plataformas, uIP es un stack TCP/IP para microcontroladores de 8 y 16 bits, con énfasis en el ahorro de recursos del sistema.

- lwIP [15]: basado en el trabajo anterior, el grupo de desarrollo de 'Light weight IP' busca disminuir aún más los requerimientos de memoria, pero manteniendo la misma funcionalidad del stack.

Es así como el programador cuenta con una amplia gama de alternativas al momento de determinar los pasos a seguir para conectar su sistema embebido a una red de área local o a Internet.

Capítulo 1
Tarjeta de Desarrollo Easyweb y su Interfaz de Red.

1.1. Introducción

La tarjeta de desarrollo Easyweb2 (fig 1.1) es adquirida por el Departamento de Electrónica de la USM a fines de 2003, con el objeto de renovar el equipamiento para los diferentes laboratorios del área de Computadores.

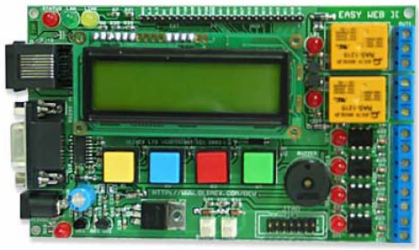


Fig. 1.1. Tarjeta de desarrollo Easyweb2

Entre sus principales características se encuentra el que cuenta con un microcontrolador MSP430, modelo F149, de la Texas Instruments; la mayor parte de los pines de este microcontrolador se hayan conectados a los numerosos periféricos presentes en la tarjeta:

- LCD de 16 x 2 caracteres.
- Cuatro pulsadores.
- Cuatro entradas optoacopladas.
- 2 salidas activadas por relé.
- Un buzzer.
- Memoria serial EEPROM 24LC515 de 64 KB de Microchip.
- 1 LED de propósito general.
- Un conector para los pines JTAG del microcontrolador.
- Salida RS232.
- Un controlador de red CS8900A de Cirrus Logic, con sus componentes externos (detalles en el punto 2 de este capítulo).

Los pines restantes se hayan asociados a los conectores 'Freq' y 'Dallas'. El primero tiene como finalidad introducir una señal de reloj externa para el Timer A del microcontrolador, y el segundo está dispuesto para la conexión de dispositivos i-Buttons, de Dallas Semiconductor. Finalmente, existen dos conectores más que permiten, por una parte, la conexión de los canales de entrada hacia el conversor análogo-digital del microcontrolador (pines que a su vez pueden ser utilizados como puertas digitales de I/O) y, por otra, el acceso a: los pines de transmisión serial, de comunicación con la EEPROM, al pin de Reset del microcontrolador, al pin P2.0 (señal auxiliar de reloj, ACLK) y a los niveles de tensión generados por los reguladores de voltaje presentes en la tarjeta para la alimentación de los integrados. Para mayores detalles acerca de la tarjeta Easyweb2, referirse a [16]. Para especificaciones del microcontrolador, referirse a [17, 18].

1.2. Interfaz de Red de la Tarjeta Easyweb2.

La descripción de la interfaz de red de la tarjeta Easyweb2 contempla el análisis de la conexión de la controladora Ethernet (Cs8900A) al microcontrolador, el detalle de la conexión de componentes adicionales, una descripción del proceso de transmisión y recepción de paquetes del Cs8900A, antecedentes acerca del packet driver a utilizar y la fundamentación de la elección del stack TCP/IP que se utilizará.

1.2.1. Conexión del chip CS8900A al MSP430F149.

El integrado CS8900A de la Cirrus Logic es una controladora LAN diseñada especialmente para aplicaciones con sistemas embebidos. Los datos se transmiten a través de un bus ISA de 16 bits, pero tiene la posibilidad de multiplexar los bytes altos y bajos, con lo que los pines necesarios para la conexión con el microcontrolador se reducen a 14: 8 pines para datos, 4 para indicar la dirección de memoria para escritura o lectura y 2 para la señalización de escritura o lectura (fig 1.2).

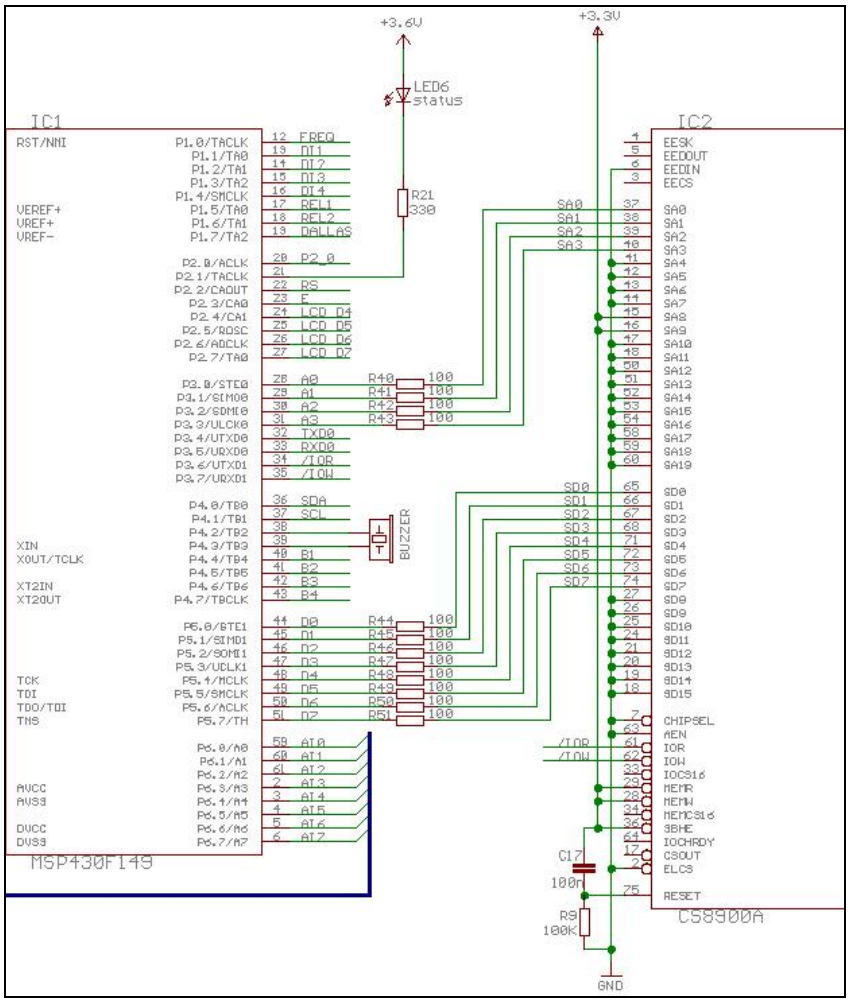


Fig. 1.2. Esquemático conexión MSPF149 a CS8900A en tarjeta Easyweb2.

Lo anterior se debe a que la controladora CS8900A puede trabajar en dos modos de operación: el primero es el modo 'memoria', en el cual los registros internos y los *buffers* para las tramas se mapean en un bloque continuo de memoria de 4 KB; la lectura y escritura se realizan seteando en cero los pines MEMR y MEMW, respectivamente. El segundo modo de operación es el 'modo de I/O' y está seteado por omisión. Para operaciones de lectura o escritura en el modo de I/O debe setearse el pin AEN en cero y la dirección de I/O de 16 bits debe coincidir con el espacio de direccionamiento del CS8900, cuya dirección base es por omisión la 0x0300. Para lectura, el pin IOR debe estar en cero. De igual modo, para escritura, el pin IOW debe estar en cero [19]. Es posible verificar a través del esquemático de la tarjeta Easyweb2 que es este último el modo escogido por diseño para la operación del chip CS8900A. El fabricante indica que no existen ventajas significativas en el desempeño de uno u otro modo de operación [20]. Ambos modos pueden operar con un bus de datos de 8 bits.

1.2.2. Conexión CS8900 a componentes externos.

El diseño integrado del chip CS8900A está pensado para minimizar la cantidad de componentes externos requeridos para establecer una conexión de red. En el caso de la tarjeta Easyweb2 se encuentran:

- 2 leds indicadores de actividad LAN y enlace de red (el driver está incorporado en el chip).
- 1 transformador de aislación de pulsos E2023 10-base T.
- 1 conector RJ-45.

Una vez incorporados estos tres componentes (fig. 1.3.), es posible conectar la tarjeta Easyweb2 a la red.

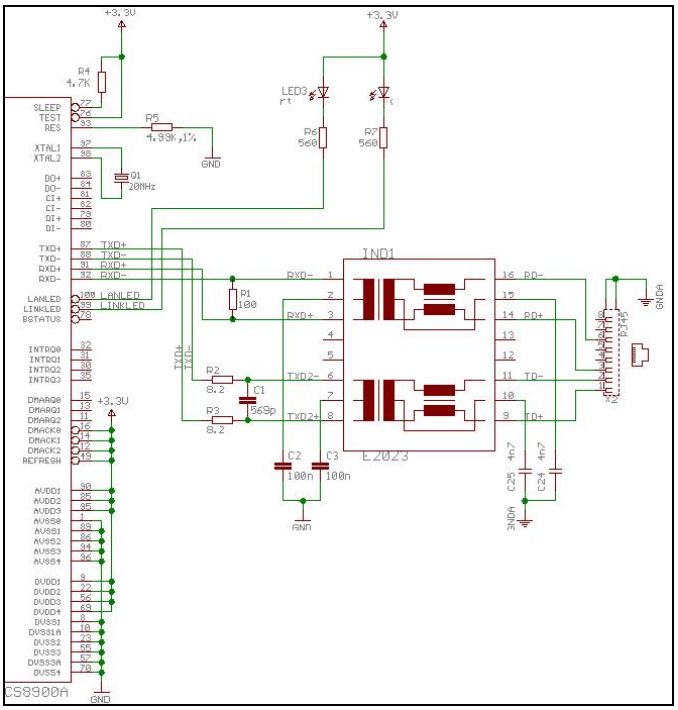


Fig. 1.3. Esquemático conexión CS8900A a componentes externos.

1.2.3. Transmisión y recepción de paquetes con el CS8900A.

Con la configuración del CS8900 ya especificada, la transmisión y recepción de paquetes se realizan en dos fases.

Para la **transmisión**, la primera fase consiste en el envío de la trama desde el **packet driver** del microcontrolador (en adelante el *host)* hacia el buffer de recepción del CS8900. Esto se inicia con el envío desde el host de un comando de transferencia, que le indica al controlador ethernet cuando iniciar la transmisión de la trama (especifica luego de cuantos bytes transferidos iniciar la transmisión) y cómo enviarla (con CRC, con bits de padding, etc.). Luego el host envía la longitud a transferir, indicando cuanto espacio se requiere en el buffer. Cuando existe espacio disponible en el buffer, el host escribe la trama ethernet en la memoria interna del CS8900A. En la segunda fase de la transmisión, el CS8900A convierte la trama en un paquete ethernet, para luego enviarla hacia la red. Esta etapa se inicia cuando ya se ha transferido el número apropiado de bytes hacia su buffer, momento en el que se transmite un encabezado y el delimitador de inicio de trama. A esto le sigue la transmisión de la dirección de destino, la dirección de origen, el campo de longitud y los datos, todos campos entregados por el host. Si la trama es menor a 64 bytes, incluyendo los bits de CRC, se incluyen los bits de padding o de extensión del paquete.

Para la **recepción**, la primera fase consiste en la llegada de un paquete desde la red y el almacenamiento de éste en el buffer de recepción del CS8900A. Esta fase se inicia cuando el paquete atraviesa el decodificador Manchester presente en el CS8900A y los datos son convertidos al formato NRZ. Luego se eliminan el encabezado y el delimitador de inicio de trama, y el resto de la trama es enviada al filtro de dirección. Si la dirección de destino es la correcta, el paquete es almacenado en la memoria interna del chip. Una vez ahí es chequeado el CRC. En la segunda fase, el host transfiere la trama recibida a través del bus de datos y lo lleva a su buffer de recepción.

Para un mayor detalle acerca de los procesos de transmisión y recepción de paquetes, referirse a [19].

1.2.4. Packet Driver para el CS8900A.

Considerando lo señalado en el punto anterior, es necesario contar con un driver que se encargue de la recepción y transmisión de los paquetes desde y hacia el controlador ethernet. Contando con el packet driver, sólo restará programar un stack TCP/IP para tener completo el software que permitirá comunicar al microcontrolador a través de la red.

Ambos programas se encuentran disponibles en la WEB, diseñados especialmente para la tarjeta de desarrollo Easyweb2. Los desarrollos corresponden a Andreas Dannenberg [21] y a Paul Curtis [22], quien portó el stack ‘uIP’ escrito por Adam Dunkels [14] a la familia de microcontroladores MSP430. Ambos programas están en lenguaje C, y fueron portados al IDE IAR Embedded Workbench para la familia de microcontroladores MSP430 v.3.1 [23], que es el ambiente de desarrollo escogido para trabajar en la programación de aplicaciones

para la tarjeta Easyweb2. Este IDE permite la fácil administración de proyectos y un ambiente de depuración en línea de los distintos desarrollos llevados a cabo, con la herramienta *Flash Emulation Tool* y el conector JTAG disponible en la tarjeta.

En lo que respecta al packet driver para el CS8900A, ambas soluciones (cs8900.c y cs8900a.c) fueron diseñadas para el modo de 8 bits en la transferencia de datos. Se implementan las siguientes funciones comunes:

- Inicialización del CS8900A[1]:

```
void cs8900a_init (void)
{
  .....

  unsigned int i;

  P3SEL = 0x30;                         // reserve P3.4 and P3.5 for rs232
  P3OUT = IOR | IOW;                    // reset outputs, control lines high
  P3DIR = 0xFF;                         // port 3 as output (all pins but rs232)

  P5SEL = 0;                            // select standard port functions
  P5OUT = 0;                            // reset outputs
  P5DIR = 0xFF;                         // switch data port to output

  Write8900(ADD_PORT, PP_SelfCTL);
  Write8900(DATA_PORT, POWER_ON_RESET);       // Reset the Ethernet-Controller

  Write8900(ADD_PORT, PP_SelfST);
  while (!(Read8900(DATA_PORT) & INIT_DONE)); // wait until chip-reset is done

  // configure the CS8900
  for (i = 0; i < sizeof InitSeq / sizeof (TInitSeq); i++)   {
    Write8900(ADD_PORT, InitSeq[i].Addr);
    Write8900(DATA_PORT, InitSeq[i].Data);
  }

  .....
}
```

Nota: los pines P5.0 a P5.7 son utilizados para la transferencia de datos, los pines P3.0 a P3.3 para el direccionamiento de lectura/escritura y los pines P3.6 y P3.7 para la señalización de lectura o escritura (fig 1.2.)

- Escritura de una palabra (16 bits) en la dirección especificada del CS8900A:

```
void
cs8900a_write(unsigned addr, unsigned int data)
{
  P5DIR = 0xFF;                             // Data port to output
  P3OUT = IOR | IOW | addr;                 // Put address on bus

  P5OUT = data;                             // Write low order byte to data bus
  P3OUT &= ~IOW;                            // Toggle IOW-signal
  P3OUT = IOR | IOW | addr | 1;             // And put next address on bus

  P5OUT = data >> 8;                        // Write high order byte to data bus
  P3OUT &= ~IOW;                            // Toggle IOW-signal
  P3OUT |= IOW;
}
```

[1] Extractos del código de cs8900a.c, con los comentarios originales del autor. Código completo en CD adjunto y en [22] .

- Lectura de una palabra desde la dirección especificada del CS8900A:

```
unsigned
cs8900a_read(unsigned addr)
{
  unsigned int value;

  P5DIR = 0x00;                          // data port to input
  P3OUT = IOR | IOW | addr;              // put address on bus

  P3OUT &= ~IOR;                         // IOR-signal low

  value = P5IN;                          // get low order byte from data bus
  P3OUT = IOR | IOW | addr | 1;          // IOR high and put next address on bus
  P3OUT &= ~IOR;                         // IOR-signal low

  value |= P5IN << 8;                    // get high order byte from data bus
  P3OUT |= IOR;

  return value;
}
```

- Envío de comandos de inicio de transferencia de datos:

```
  // Transmit command
  cs8900a_write(TX_CMD_PORT, TX_START_ALL_BYTES);
  cs8900a_write(TX_LEN_PORT, framesize);


  Nota: framesize corresponde a una variable del stack TCP/IP que indica la
  longitud de la trama a ser transferida.
```

- Verificación de espacio en buffer de recepción del CS8900A:

```
for (;;)
    {
      // Check for avaliable buffer space
      cs8900a_write(ADD_PORT, PP_BusST);
      if (cs8900a_read(DATA_PORT) & READY_FOR_TX_NOW)
        break;

      .....
    }

Nota: al existir espacio en buffer, se ejecuta la transferencia de datos hacia el
CS8900A.
```

- Detección de la llegada de un frame:

```
// Check receiver event register to see if there are any valid frames avaliable
  cs8900a_write(ADD_PORT, PP_RxEvent);
  if ((cs8900a_read(DATA_PORT) & 0xd00) == 0)   return 0;
```

A diferencia del driver original `cs8900.c` , el driver `cs8900a.c` contempla la invocación desde el programa principal de sólo tres funciones, las que durante su ejecución invocan a todas las demás:

- `cs8900a_init()`, que como ya se vio inicializa el controlador ethernet; además configura el módulo de reloj del microcontrolador, seteando los osciladores; por último, configura el Timer A para generar interrupciones periódicas, que implementan a través de la rutina de servicio presente en este mismo driver, un reloj de tiempo real.

- `cs8900a_send()`, que contempla el envío de los comandos que inician la transferencia, la espera por espacio en buffer y finalmente la transferencia propiamente tal.
- `cs8900a_poll()`, que implementa la revisión de la llegada de nuevos paquetes, el descarte de aquellos que sean muy grandes para el buffer de recepción del stack TCP/IP y finalmente la recepción de los datos.

Nota: esta revisión deber ser realizada periódicamente; lo ideal sería que los eventos de transmisión y recepción de paquetes fueran atendidos por rutinas de servicio de interrupciones, pero esto no es factible por dos motivos: el controlador cs8900A no soporta interrupciones en el modo de 8 bits y las puertas digitales del microcontrolador susceptibles de gatillar interrupciones son solamente la 1 y la 2.

En resumen, el driver `cs8900a.c` programado para su uso con el stack TCP/IP de Adam Dunkels 'uIP', constituye una optimización del packet driver original `cs8900.c`, y es la alternativa a utilizar si la elección de stack corresponde a 'uIP'.

1.2.5. Stack TCP/IP.

Como se mencionó en el punto 1.2.4., luego de tener un packet driver funcional sólo resta contar con un stack TCP/IP, es decir, la implementación de los protocolos mínimos para poder comunicar al microcontrolador a través de la red.

Se trabajó con dos stack TCP/IP: en una primera etapa de familiarización con la tarjeta, se ocupó el stack programado pensando específicamente en la tarjeta Easyweb2, contenido principalmente en el código fuente `tcpip.c` (*Dannenberg*), y disponible en la WEB en [21]. En una segunda etapa, se utilizó el stack 'uIP' (*Dunkels*), que fue portado para la familia de microcontroladores MSP430 por Paul Curtis, e implementado en un proyecto disponible en la WEB en [22] para el compilador Crossworks de Rowley Associates.

El stack TCP/IP de Andreas Dannenberg fue diseñado fundamentalmente para montar un pequeño servidor web en la tarjeta Easyweb2 y por esto cuenta con una serie de limitaciones:

- Soporta sólo una conexión TCP/IP. Cualquier otro requerimiento de conexión es descartado por el stack.
- No tiene verificación de checksum, por lo que no existe ningún tipo de detección de errores en los paquetes.
- No existe procesamiento de paquetes previamente fragmentados.
- No tiene soporte para datagramas UDP.

Todas estas limitaciones son superadas por 'uIP', a la vez que entrega una API estructurada y más simple de utilizar a la hora de crear aplicaciones propias, gracias también a su excelente documentación y a los ejemplos proporcionados.

Es así como 'uIP' es el stack TCP/IP escogido para el desarrollo de aplicaciones de red a lo largo de este trabajo.

Capítulo 2
Uso de la API del stack uIP.

Antes de programar una primera aplicación es necesario conocer la API de uIP. El punto siguiente constituye un resumen de la descripción de la API que se encuentra en la documentación de uIP [14]:

2.1. API (*Application Program Interface*)

La API de uIP define el modo mediante el cual la aplicación interactúa con las funciones del stack. Como se vio en la introducción, la API más utilizada para TCP/IP es la API mediante sockets de BSD, utilizada en la mayor parte de los sistemas UNIX y que influenció fuertemente la API de Windows, Winsock. Debido a que esta API utiliza semántica de *stop and wait*, requiere el soporte de un sistema operativo multitarea. Pero uIP está orientado al ahorro de espacio en memoria y no contempla de antemano el uso de un sistema operativo en tiempo real, por lo que se utiliza un enfoque distinto al de los sockets de BSD.

uIP utiliza una interfaz mediante la cual la aplicación es invocada en respuesta a ciertos eventos. Para esto, la función debe ser implementada como una función en C que corresponda a la macro `UIP_APPCALL()`; esta función será invocada por uIP cada vez que ocurra el evento respectivo. Cada evento tiene asociado una función que revisa su ocurrencia y retorna un valor igual a cero en caso negativo o distinto de cero en caso que se produzca el evento. La siguiente es una breve descripción de algunos de los posibles eventos, con su función respectiva:

- **Recepción de datos**: si la función `uip_newdata()` retorna un valor diferente a cero, significa que el host remoto ha enviado nuevos datos. El tamaño de los datos se obtiene a través de la función `uip_datalen()`. Es tarea de la aplicación utilizar estos datos directamente, sin almacenarlos, o bien copiarlos a un buffer propio, para procesarlos.

- **Envío de datos**: la aplicación envía datos a través de la función `uip_send()`. Esta función toma dos argumentos: un puntero a los datos que se enviarán y su longitud. Si la aplicación requiere espacio en RAM para generar los datos que se pretende enviar, es posible utilizar el packet buffer, que es apuntado por el puntero `uip_appdata`. No es posible invocar a `uip_send()` más de una vez por llamado a la aplicación.

- **Establecimiento de una nueva conexión**: la función encargada de abrir nuevas conexiones es `uip_connect()`. Esta función configura la conexión en el estado SYN_SENT, y setea el timer de retransmisiones en cero; el segmento TCP de tipo SYN es enviado en la siguiente oportunidad en que se realice el procesamiento periódico de conexiones (0,5 [s] después de la invocación de `uip_connect()` [14]). Esta función retorna un puntero a la estructura `uip_conn`

para la nueva conexión, o bien NULL si no fue posible establecer la conexión. La estructura `uip_conn` puede ser usada para distinguir entre diferentes servicios, a través del número de puerto de la conexión, o bien para verificar la dirección IP a la que se está conectado.

- **Escuchar puertos**: una forma pasiva de establecer una conexión es invocar a la función `uip_listen()`. Si llega una petición de conexión a un puerto que se ha dejado escuchando, uIP crea una nueva conexión e invoca a `UIP_APPCALL()`. La función `uip_connected()` retorna un valor distinto de cero si la aplicación fue invocada debido a que se creó una nueva conexión. Puede verificarse el campo `lport` en la estructura `uip_conn` para determinar a través de cual puerto se realizó la conexión.

- **Cerrar conexiones**: la aplicación cierra la conexión mediante la invocación de `uip_close()`, durante el llamado a UIP_APPCALL(). Si se requiere indicar la ocurrencia de un error, puede invocarse a `uip_abort()`. Si la conexión fue terminada por el host remoto, la función `uip_closed()` retorna un valor diferente a cero.

En lo que respecta al driver del dispositivo de red, en este caso `cs8900a.c`, la interacción con uIP se limita a dos funciones: `uip_input` y `uip_periodic()`.La primera función debe ser invocada cuando se ha recibido un paquete IP y se le ha colocado en el packet buffer `uip_buf`. Esta función procesa el paquete y cuando retorna puede haber colocado un paquete de respuesta en el mismo buffer (lo que es indicado por la variable `uip_len` > 0). Luego de esto, el driver puede enviar este último paquete hacia la red. La segunda función, `uip_periodic()`, es invocada periódicamente y una vez por cada conexión establecida (idealmente cada un segundo). Esta función es utilizada por uIP para manejar los timers que regulan los protocolos; cuando retorna, puede haber colocado un paquete en el buffer `uip_buf`, y el driver debe enviarlo.

2.2. Desarrollo de la primera aplicación.

Como se señaló en el capítulo anterior, el stack uIP fue portado a la familia de microcontroladores MSP430 por Paul Curtis, quien utilizó el compilador Cross Studio de Rowley Associates; este proyecto se encuentra en [22]. Este código fue a su vez portado por Leopoldo Silva B. para el compilador de IAR Systems, Embedded Workbench v3, que es el ambiente de desarrollo utilizado en este trabajo. Para la creación de un *workspace* y de un proyecto con este compilador, y para familiarizarse con las herramientas de depuración que ofrece, consultar [24].

Los archivos del stack uIP que deben incluirse al proyecto para desarrollar una primera aplicación se observan en la siguiente figura (fig. 2.1.):

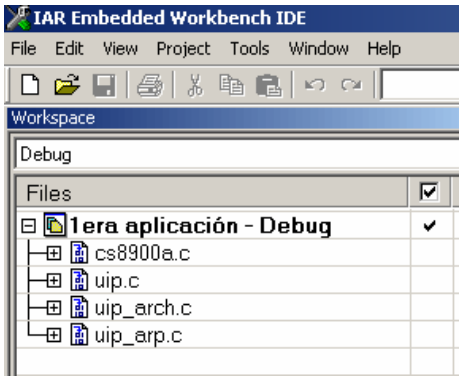


Fig. 2.1. Archivos uIP para desarrollo de una aplicación de red.

De estos archivos, `uip.c` contiene el código del stack TCP/IP propiamente tal, `uip_arch.c` contiene las funciones que implementan el algortimo de checksum y la suma de campos de 32 bits y `uip_arp.c` contiene la implementación del protocolo ARP. Ya se ha señalado que `cs8900a.c` corresponde al driver del controlador ethernet y debe incluirse también en el proyecto.

El archivo `uip_opt.h` no debe añadirse al proyecto, pues es incluido por el código `uip.c` y por su archivo de definiciones `uip.h`. Este archivo incluye las opciones de configuración del stack TCP/IP: configuración de la dirección IP y de la máscara de su-red del host, y de la dirección IP de la puerta de enlace pre-determinada. Se incluye aquí, además, el archivo de definiciones de la aplicación a ser incorporada al proyecto.

Sólo resta por incluir al proyecto el loop de control principal, usualmente denominado `main.c` y los archivos que contienen el código de la aplicación.

Con respecto a `main.c`, corresponde al código que invoca a las funciones uIP en un orden que permite procesar los requerimientos de una conexión TCP. Junto a la distribución de uIP, se entregan varias aplicaciones, cada una con su propio loop de control principal. Desde la aplicación que implementa un servidor web, se toma el archivo `main.c` para utilizarlo en la primera aplicación a desarrollar. Este loop de control se resume en el siguiente diagrama (fig. 2.2.):

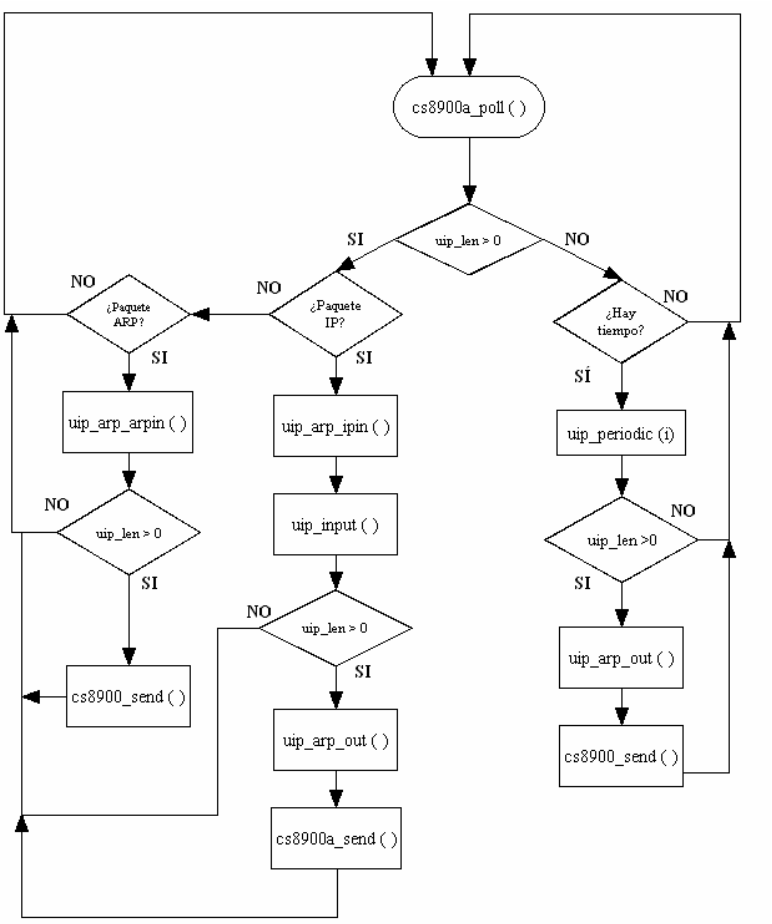


Fig. 2.2. Diagrama del loop de control de la aplicación.

Como se observa en el diagrama, se está constantemente revisando la llegada de algún paquete mediante la función `cs8900a_poll()`. Esta función retorna el tamaño del paquete, si es que hay alguno presente. Este valor se almacena en la variable `uip_len` y a través de ella el stack determina si debe comenzar a procesar el paquete o bien realizar las funciones periódicas.

Para determinar si corresponde a un paquete IP o no, se utiliza el siguiente código[2]:

```
if(BUF->type == htons(UIP_ETHTYPE_IP)) {
      ....                                    // procesa paquete IP
}
```

2.2.1. Recepción y envío de un paquete IP

Si se cumple la condición anterior, se invoca a la función `uip_arp_ipin()`. Esta función debe ser llamada cuando se recibe un paquete IP, ya que verifica si la dirección de origen del paquete se encuentra en el caché ARP de uIP. Si es así, actualiza la información; en caso contrario, crea una nueva entrada en la tabla.

[2] Extractos del código main.c, correspondiente al web server incluído con uIP. Código completo en CD adjunto y en [22].

Posteriormente, es invocada `uip_input()`, función encargada de procesar el paquete entrante. Este paquete debe estar presente en el buffer `uip_buf` y la longitud del paquete debe estar contenida en la variable `uip_len`. Al retorno de esta función puede que se haya colocado un paquete de salida en `uip_buf`. Para saberlo, se verifica nuevamente el valor de la variable uip_len. Si es diferente de cero, quiere decir que existe un paquete por enviar, para lo cual se invoca a `uip_arp_out()`. Esta función debe ser llamada antes de enviar un paquete IP, ya que verifica la dirección IP de destino del paquete para saber que dirección MAC utilizar. Si la dirección IP en cuestión se encuentra en la red de área local, se la busca en la tabla ARP. Si existe una entrada que coincida, se prepara el encabezado ethernet y la función retorna. En caso contrario, el paquete presente en el buffer `uip_buf` es reemplazado por una petición ARP para la dirección IP en cuestión. El paquete IP es descartado y se asume que protocolos de capa superior (por ejemplo TCP), se encargarán de la retransmisión del paquete en cuestión. Por último, si la dirección IP no se encuentra en la red de área local, se utiliza la dirección IP de la puerta de enlace pre-determinada.

Al retorno de la función `uip_arp_out()`, existe un paquete presente en `uip_buf` y su tamaño se encuentra en `uip_len`. Con estas variables adecuadamente preparadas, el driver del controlador ethernet envía el paquete mediante la función `cs8900a_send()`.

2.2.2. Recepción y envío de un paquete ARP

Si el paquete no es IP, se verifica que sea de tipo ARP, mediante el siguiente código[1]:

```
if(BUF->type == htons(UIP_ETHTYPE_ARP)){
  ....                                  // procesa paquete ARP
}
```

Al cumplirse la condición anterior, se ha recibido un paquete ARP. Este puede ser de dos tipos: una respuesta a una petición ARP anterior, o bien una petición ARP entrante. En el primer caso se toman los datos recibidos y se les incluye en la tabla ARP. Si se trata de una petición por la dirección IP del host uIP, se crea un paquete de respuesta y se le coloca en el buffer `uip_buf`. El tamaño del paquete queda almacenado en `uip_len`, lo que permite determinar si debe ser enviado mediante la función `cs8900a_send()`.

2.2.3. Procesamiento periódico

De no existir un paquete presente en el buffer del controlador ethernet, y si se cuenta con el tiempo para ello, se realiza el procesamiento periódico. Esta última condición se verifica del siguiente modo[3]:

```
start = rt_ticks;          //inicialización de main.c

while (1) {
        ...

       current = rt_ticks;
```

[3] Extracto del código main.c, correspondiente al web server incluído con uIP. Código completo en Anexo.

```
        if((u16_t)(current - start) >= (u16_t)RT_CLOCK_SECOND/2) {
              ...           // procesamiento periódico
        }
          ...
}
```

La macro `RT_CLOCK_SECOND` debe corresponder al número aproximado de ticks (depende de la inicialización de los osciladores) que transcurren en un segundo. Esto da cuenta de la necesidad de incorporar tiempo real al sistema a través de timers.

Si la condición se cumple, se realiza el procesamiento mediante la función `uip_periodic()`. Al retorno de esta función puede existir un paquete presente en uip_buf. Si esto es así, es necesario enviar un paquete, lo que se realiza mediante el llamado a `uip_arp_out()` y `cs8900a_send()`, por las razones ya descritas.

2.2.4. Inicialización

Además del loop ya descrito, es necesario realizar las siguientes inicializaciones en `main.c`:

```
uip_init();               // inicialización del stack TCP/IP

uip_arp_init();           // inicialización del módulo ARP

cs8900a_init();           // inicialización controlador ethernet

app1_init();       // inicialización de la aplicación
```

Con respecto a la aplicación, ésta consiste en dos archivos, uno con el código de la aplicación propiamente tal (`app1.c`) y otro que contenga como mínimo las definiciones de las estructuras de datos que se requieran (usualmente una estructura que implemente una máquina de estados para la aplicación), la inclusión del archivo `uip_opt.h` y la definición de la función principal de la aplicación como `UIP_APPCALL()` (`app1.h`).

La primera aplicación desarrollada consiste en un pequeño ejemplo de servidor TCP, que abre pasivamente una conexión mediante la función `uip_listen()`; es decir, queda a la espera de conexiones entrantes por el puerto que se especifique.

La función `UIP_APPCALL()`, invocada periódicamente por uIP, tan sólo verifica si han llegado datos provenientes del host remoto; si es así, envía mediante la función `uip_send()` el mensaje “msp430”.

Los códigos de `app1.h` y `app1.c` son los siguientes:

```
//*************app1.h*********************

#include "uipopt.h"

void app1_init(void);
void app1_appcall(void);

#ifndef UIP_APPCALL
#define UIP_APPCALL     app1_appcall
#endif


//*************app1.c*********************

#include "uip.h"
#include "ej1.h"

#include <string.h>

void app1_init(void) {
   uip_listen(HTONS(1234));
}

void app1_appcall(void) {
   if( uip_newdata() ) {
      uip_send("msp430\n", 7);
   }
}
/*****************************************
```

Cabe reiterar que en el archivo `uip_opt.h` es necesario incluir el archivo de definiciones de la aplicación, en este caso `app1.h`. Debe a su vez configurarse la dirección IP del host y de la puerta de enlace pre-determinada.

Con lo anterior debidamente programado, basta incluir los archivos `main.c` y `app1.c` al proyecto en cuestión, compilarlo y descargarlo a la memoria de programa del microcontrolador y finalmente conectarlo a la red, a través del puerto RJ-45 presente en la tarjeta, para verificar el correcto funcionamiento de la aplicación.

2.3. Conexión a la red y configuración de cliente TCP/IP.

Un esquema de la conexión realizada es el siguiente (fig. 2.3.):

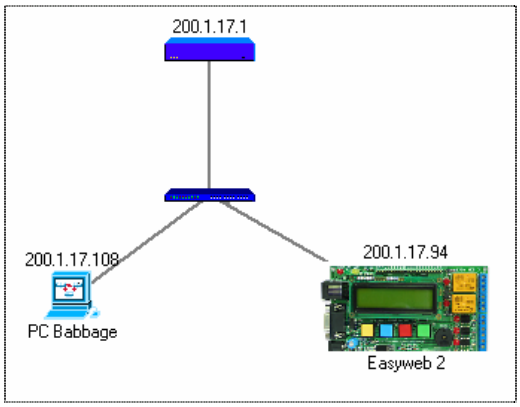


Fig. 2.3. Conexión de tarjeta Easyweb2 a la red.

Gracias a esta configuración, y con la interfaz de red del PC en modo promiscuo, es posible analizar mediante un 'sniffer' el tráfico entre la aplicación que corre en la tarjeta y la red de área local.

El stack uIP incluye una implementación del protocolo ICMP, por lo que puede efectuarse un *ping* desde una máquina presente en la sub-red para probar conectividad. En la siguiente figura, se observa la respuesta de la tarjeta a un *ping* efectuado desde la línea de comandos de Windows en PC Babbage (fig. 2.4.):

No.	Time (h:m:s:ms)	MAC source addr	MAC dest. addr	Frame	Protocol	Addr. IP src	Addr. IP dest	Port src	Port dest
0	11:0:29:054	00-05-5D-4D-1F-A5	FF-FF-FF-FF-FF-FF	ARP	ARP->Request	No IP address	No IP address	---	---
1	11:0:29:054	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	ARP	ARP->Reply	No IP address	No IP address	---	---
2	11:0:29:054	00-05-5D-4D-1F-A5	00-BD-3B-33-05-79	IP	ICMP->Echo_Request	200.1.17.108	200.1.17.94	---	---
3	11:0:29:054	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	IP	ICMP->Echo_Reply	200.1.17.94	200.1.17.108	---	---
4	11:0:30:054	00-05-5D-4D-1F-A5	00-BD-3B-33-05-79	IP	ICMP->Echo_Request	200.1.17.108	200.1.17.94	---	---
5	11:0:30:054	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	IP	ICMP->Echo_Reply	200.1.17.94	200.1.17.108	---	---
6	11:0:31:054	00-05-5D-4D-1F-A5	00-BD-3B-33-05-79	IP	ICMP->Echo_Request	200.1.17.108	200.1.17.94	---	---
7	11:0:31:054	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	IP	ICMP->Echo_Reply	200.1.17.94	200.1.17.108	---	---
8	11:0:32:054	00-05-5D-4D-1F-A5	00-BD-3B-33-05-79	IP	ICMP->Echo_Request	200.1.17.108	200.1.17.94	---	---
9	11:0:32:054	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	IP	ICMP->Echo_Reply	200.1.17.94	200.1.17.108	---	---

Fig. 2.4. Prueba de conectividad mediante ICMP.

En la figura anterior es posible apreciar dos paquetes ARP. Uno corresponde a la petición ARP con dirección destino de broadcast efectuada por el cliente, con el fin de obtener la dirección MAC del servidor. El otro es la respuesta ARP dirigida por el servidor con la información que solicitó el cliente. Los demás paquetes corresponden a las peticiones y respuestas ICMP.

Cabe hacer notar que el analizador de paquetes entrega la información de las direcciones MAC e IP tanto de origen como destino, además del protocolo al que corresponden los paquetes.

2.4. Conexión con un cliente TCP/IP.

Para establecer una conexión con la aplicación de servidor de la tarjeta, se requiere un cliente TCP que se conecte a través del puerto en que está escuchando la aplicación.

2.4.1. Cliente TCP/IP en Windows.

Una opción es utilizar la Hyper-terminal de Windows, en modo TCP/IP (fig. 2.5.):

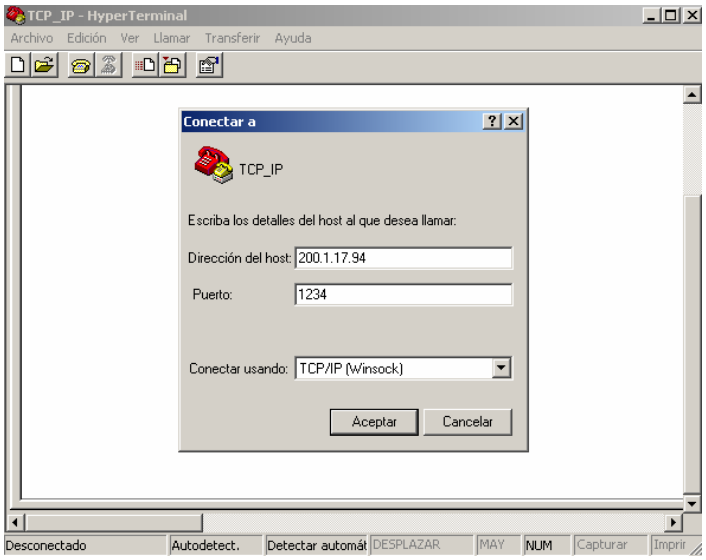


Fig. 2.5. Cliente TCP/IP Winsock.

Al hacer click en Aceptar, se efectúa la conexión TCP/IP entre el cliente (200.1.17.108) y el servidor (200.1.17.94). El detalle de los paquetes capturados por el analizador de paquetes al momento del establecimiento de la conexión se observa en la siguiente figura (fig. 2.6.):

No.	Time (h:m:s:ms)	MAC source addr	MAC dest. addr	Frame	Protocol	Addr. IP src	Addr. IP dest	Port src	Port dest	SEQ	ACK	Size
0	10:38:25:695	00-05-5D-4D-1F-A5	FF-FF-FF-FF-FF-FF	ARP	ARP->Request	No IP address	No IP address	---	---			42
1	10:38:25:695	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	ARP	ARP->Reply	No IP address	No IP address	---	---			60
2	10:38:25:695	00-05-5D-4D-1F-A5	00-BD-3B-33-05-79	IP	TCP	200.1.17.108	200.1.17.94	4520	1234	1905401935	0	62
3	10:38:25:695	00-BD-3B-33-05-79	00-05-5D-4D-1F-A5	IP	TCP	200.1.17.94	200.1.17.108	1234	4520	1470	1905401936	60
4	10:38:25:695	00-05-5D-4D-1F-A5	00-BD-3B-33-05-79	IP	TCP	200.1.17.108	200.1.17.94	4520	1234	1905401936	1471	54

Fig. 2.6. Tráfico de paquetes en establecimiento de conexión TCP/IP.

Los dos primeros paquetes corresponden al protocolo ARP y ya han sido descritos. Los siguientes tres paquetes corresponden a la negociación TCP/IP y dan cuenta de que la conexión ha sido establecida. De este modo, cualquier caracter que se envíe a través de la terminal es considerado como nuevos datos por el servidor, que responde en cada ocasión con el string antes señalado (fig. 2.7.):

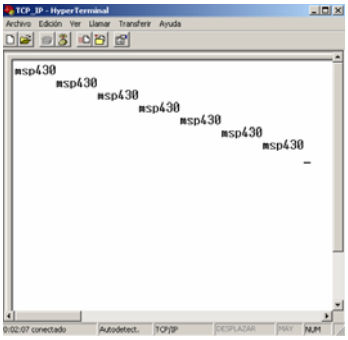


Fig. 2.7. Conexión TCP/IP establecida e intercambio de datos.

Otra posibilidad es utilizar la línea de comandos, abriendo una sesión telnet a través del puerto especificado anteriormente mediante la sintaxis: telnet <IPhost> <Número de puerto>.

2.4.2. Programación de un Cliente TCP/IP.

Una alternativa al empleo de un cliente ya existente corresponde a la programación de un cliente TCP/IP. Para este efecto, y a lo largo de todo este trabajo, se ha escogido el lenguaje Java ya que constituye una herramienta que permite a cualquier programador (sin necesidad de conocimientos avanzados de comunicaciones) escribir programas que se integren fácilmente en una red IP para acceder a bases de datos remotas, interactuar con otros programas (escritos o no en Java) y distribuir datos o aplicaciones a través de Internet [25].

Para programar un cliente TCP en Java, es conveniente familiarizarse con el concepto de *stream*, o flujo. Un stream puede considerarse como una cañería en la que es posible leer o escribir bytes. No importa lo que pueda haber en el otro extremo del tubo: un teclado, un monitor, un archivo, un proceso, un objeto Java o una conexión TCP/IP. Todos los streams que se utilizan en Java (reunidos generalmente en el paquete `java.io.*`) pertenecen a dos clases abstractas comunes: `java.io.InputStream` para los flujos de entrada, a partir de los cuales es posible leer, y `java.io.OutputStream` para los flujos de salida, que son aquellos utilizados para escribir.

Las funciones de red, en este caso la creación de sockets, se encuentran en el paquete `java.net.*`. Para crear un socket, debe especificarse la IP y el número de puerto a través del cual abrir la conexión TCP/IP.

El cliente a programar requerirá establecer la conexión, enviar un string cualquiera hacia el servidor y recibir y desplegar el string de respuesta ya conocido. El cliente programado se resume en el siguiente esquema (fig. 2.8.):

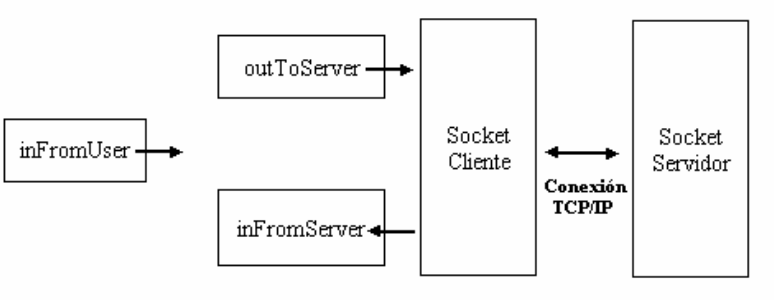


Fig. 2.8. Streams y socket de cliente TCP/IP.

El stream `inFromUser` pertenece a la clase `BufferedReader`, que permite leer texto desde un stream de entrada de caracteres. Al inicializarla, se la asocia a la entrada estándar (`system.in`, el teclado), incorporando la conversión de bytes a caracteres mediante la clase `InputStreamReader`; esto permite recoger los caracteres ingresados por el usuario, que se almacenan como los strings `serverHost` (para configurar IP del servidor), `serverPort` (número de puerto), o `line`, dependiendo de la etapa del programa.

El stream `inFromServer` se crea de un modo similar al anterior, pero al inicializarlo, se la asocia al stream de entrada del socket cliente.

Finalmente, el stream `outToServer` pertenece a la clase `DataOutputStream`. Cualquier secuencia de caracteres escrito en este stream fluirá a través del socket cliente por la cañería TCP (proceso transparente para el usuario).
El resto del código es autoexplicativo, y es el siguiente:

```
/*Cliente TCP/IP
******************/
import java.io.*;
import java.net.*;

public class TCPclient {
  public static void main(String[] args)
    throws Exception
  {
    BufferedReader inFromUser =
      new BufferedReader(new InputStreamReader(System.in));

    System.out.print("Ingrese IP del servidor: ");
    System.out.flush();

    String serverHost = inFromUser.readLine();

    System.out.print("Ingrese número de puerto: ");
    System.out.flush();

    int serverPort = Integer.parseInt(inFromUser.readLine());

    /* creación socket - se establece conexión TCP/IP */
    Socket clientSocket = new Socket(serverHost, serverPort);

    DataOutputStream outToServer =
      new DataOutputStream(clientSocket.getOutputStream());
    BufferedReader inFromServer =
      New BufferedReader(new InputStreamReader( clientSocket.getInputStream()));

    System.out.print("Cliente: ");
    System.out.flush();
    String line = inFromUser.readLine();              // lee una línea hasta '\n'

    while (!line.equals("exit")) {
      outToServer.writeBytes(line + '\n');

      String uIPmsg = inFromServer.readLine();
```

```
      System.out.println("uC: " + uIPmsg);

      System.out.print("Cliente: ");
      System.out.flush();
      line = inFromUser.readLine();
    }

    clientSocket.close();                              // cierra socket: termina conexión
  }
}
```

La ejecución de este cliente se realiza a través de línea de comando (la terminal de I/O del IDE Netbeans 3.6) y el resultado se aprecia en la figura (2.9.):

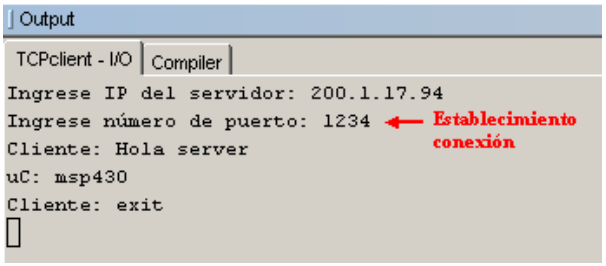


Fig. 2.9. Ejecución del cliente “TCPclient”.

El análisis de los paquetes puede realizarse de un modo similar al ya descrito.

Capítulo 3
Diseño estructurado de aplicaciones.

Para la programación de aplicaciones más complejas, es recomendable utilizar la API de uIP de una manera estructurada.

En el capítulo anterior se veía cómo una aplicación puede establecer una conexión y realizar el envío de una misma secuencia de datos ante la ocurrencia de cualquier evento. Esto mostró la capacidad de conectividad del stack.

3.1. Administrador de eventos.

Si lo que se desea es implementar un protocolo de comunicación más elaborado, en el que la transferencia de datos entre el cliente y el servidor dependa del estado en que se encuentre la conexión y del tipo y cantidad de datos ya transferidos, es útil pensar la aplicación como un administrador de eventos, que discrimine y actúe ante la ocurrencia de diferentes sucesos.

Este administrador de eventos será la función `UIP_APPCALL()`, que deberá ser capaz de:

- Administrar el establecimiento de las conexiones.
- Administrar el proceso de recepción de nuevos datos.
- Enviar datos.
- Actuar ante la confirmación de que los datos enviados han sido recibidos exitosamente por el host remoto (*acknowledge*)
- Manejar las situaciones de error.

Una alternativa de implementación de esta función es la siguiente [14]:

```
#define UIP_APPCALL     eventhandler_app

void eventhandler_app(void) {
        if(uip_aborted()) {
          aborted();
        }
        if(uip_timedout()) {
          timedout();
        }
        if(uip_closed()) {
          closed();
        }
         if(uip_connected()) {
          connected();
        }
        if(uip_acked()) {
          acked();
        }
        if(uip_newdata()) {
          newdata();
          senddata();
        }
}
```

Lo primero que realiza esta función es manejar las situaciones de error que pudieran producirse durante la conexión. Para esto, verifica si `uip_aborted()` y `uip_timedout()` retornan un valor diferente de cero. Si éste fuera el caso, las funciones `aborted()` y `timedout()` deberán programarse para realizar la acción deseada. De la misma manera, si la conexión ha sido terminada, `uip_closed()` devolverá un valor distinto a cero y la función `closed()` deberá ser programada para manejar la situación.

Posteriormente, se revisa si existe una nueva conexión establecida, caso en el que la función `uip_connected()` retorna un valor diferente a cero. La función `connected()` deberá ser programada para realizar lo que se necesite hacer una vez que se ha establecido la conexión.

Del mismo modo, las funciones `uip_acked()` y `uip_newdata()` permiten verificar si existe recepción de un ACK o de nuevos datos, respectivamente. Las funciones `acked()`, `newdata()` y `senddata()` deberán programarse para tomar alguna decisión, realizar el procesamiento de los nuevos datos recibidos y enviar datos hacia el host remoto, respectivamente.

3.2. Programación de una aplicación estructurada (`app2.c`)

Para ejemplificar lo expuesto anteriormente, se programó una aplicación simple que responde al siguiente diagrama de estados (fig 3.1.):

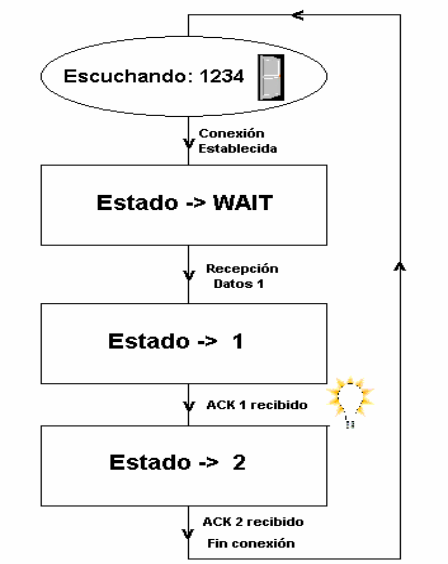


Fig. 3.1. Diagrama de estados de la aplicación.

La inicialización de la aplicación, `app2_init()`, abre pasivamente una conexión en el puerto 1234, y configura el pin P2.1 para operación como puerta digital de salida. De esta manera, es posible manejar el led de estado (encenderlo o apagarlo) de la tarjeta de desarrollo Easyweb2.

Esta aplicación utilizará la estructura de datos `app2_struct` (fig. 3.2.):

Fig. 3.2. Esquema estructura `app2_struct`.

Esto permite, a través de la variable state (`u8_t: typedef unsigned char`) implementar la máquina de estados propuesta en la fig. 3.1.; se cuenta además con los campos `textptr` y `textlen`, para pasar a la función `uip_send()` un puntero a los datos a enviar y su longitud, respectivamente.

Esta aplicación no contará con un manejo de errores en la conexión, por lo que las funciones `aborted()`, `timedout()` y `closed()` se declaran vacías.

La función `connected()` es invocada cuando se ha establecido una conexión a través del puerto 1234. Su acción es la de dejar a la aplicación en el estado cero, o WAIT. Asigna además el valor cero al campo `textlen` (no hay datos que enviar aún).

La primera vez que llegan datos desde el host remoto, la función `newdata()` encuentra a la aplicación en el estado WAIT. Su acción en este caso es la preparación del mensaje: `"MSP430: Host remoto x.x.x.x conectado"`, donde el campo `x.x.x.x` es completado con la dirección IP del host remoto, obtenida a través de la estructura `uip_conn` en su campo `ripaddr` (`uip_conn->ripaddr`). Este mensaje es almacenado en el arreglo de caracteres `mes1[]` y se le pasa al stack para el posterior envío, a través del puntero `s`, a la estructura `app2_struct` (campos `textptr` y `textlen`). Por último, se deja la aplicación en el estado 1.

La función `senddata()` administra el envío de datos, ejecutando la función `uip_send()` si el campo `textlen` contiene un valor diferente a cero. Los datos se envían, y el si host remoto los recibe adecuadamente, envía el ACK correspondiente. Al recibir este ACK, mediante la función `acked()`, la aplicación se encuentra en el estado 1. En primer lugar, se actualizan los campos `textlen` y `textptr` con la información contenida en `uip_len`, que indica la longitud de los datos efectivamente enviados. Si se ha transmitido el mensaje completo, el campo `textlen` será nuevamente cero y se ejecutará la acción correspondiente al estado 1, que es avanzar al estado 2. Además, como indicación de que el primer ACK fue recibido, se enciende el led de estado de la tarjeta Easyweb2.

La aplicación continuará en el estado 2 hasta que lleguen nuevos datos desde el host remoto. En este caso, la función newdata() pasa el mensaje de cierre de conexión: “MSP430: Conexión Terminada”. Si el host remoto recibe correctamente el mensaje, envía un segundo ACK, que es procesado nuevamente por la función `acked()`. Como la aplicación se encuentra ahora en el estado 2, la acción ejecutada es apagar el led de estado, y cerrar la conexión mediante `uip_close()`.

El código de la aplicación es el siguiente:

```
//*************app2.h*********************
#include "uipopt.h"


static void aborted(void);
static void timedout(void);
static void closed(void);
static void connected(void);
static void newdata(void);
static void acked(void);
static void senddata(void);

void call_app2(void);
void app2_init(void);


struct app2_struct {
  u8_t state;
  char *textptr;
  int  textlen;
};

#ifndef UIP_APPCALL
#define UIP_APPCALL     call_app2
#endif

#ifndef UIP_APPSTATE_SIZE
#define UIP_APPSTATE_SIZE (sizeof(struct app2_struct))
#endif


//*************app2.c*********************

#include "uip.h"
#include "app2.h"
#include <msp430x14x.h>
#include <string.h>

#define STATE_WAIT    0
#define STATE_1       1
#define STATE_2       2

char mes1[50];
char mes2[27] = "MSP430: Conexion Terminada\n";


void app2_init(void) {
   uip_listen(HTONS(1234));
   P2SEL = 0;
   P2DIR |= BIT1;
   P2OUT |= BIT1;
}

/*Administrador de eventos*/
void call_app2(void) {
  if(uip_aborted()) {
    aborted();
  }
  if(uip_timedout()) {
    timedout();
  }
  if(uip_closed()) {
    closed();
  }
  if(uip_connected()) {
    connected();
  }
  if(uip_acked()) {
    acked();
```

```
  }
  if(uip_newdata()) {
    newdata();
    senddata();
  }
}

/*No existe control de errores en la conexión*/
static void aborted(void) {}
static void timedout(void) {}
static void closed(void) {}


static void connected(void) {
  /*se define puntero s local a la función, que apunta a la estructura uip_conn (propia del
   stack y almacenada en RAM), campo appstate para administrar el estado de la aplicación y
   manejar el paso de datos hacia el host remoto con un uso eficiente de memoria */

  struct app2_struct *s = (struct app2_struct *)uip_conn->appstate;

  /*Inicialización máquina estados*/
  s->state   = STATE_WAIT;
  s->textlen = 0;
}

static void newdata(void) {
  /*Buffers para ip host remoto*/
  u16_t iprhost [2];
  u16_t iprh[4];

  struct app2_struct *s = (struct app2_struct *)uip_conn->appstate;

  switch (s->state) {

   case STATE_WAIT:
    iprhost[0]= *(uip_conn->ripaddr);
    iprhost[1]= *(uip_conn->ripaddr+1);

    /*Obtención ip host remoto*/
    iprh[1]= iprhost[0]>>8;
    iprh[0]= ((iprhost[0]<<8)>>8);
    iprh[3]= iprhost[1]>>8;
    iprh[2]= ((iprhost[1]<<8)>>8);

    /*Preparación mensaje 1*/
    sprintf(mes1,"MSP430: Host remoto %d %d %d %d conectado\n",iprh[0],iprh[1],
    iprh[2],iprh[3]);
    s->textptr = mes1;
    s->textlen = sizeof(mes1);
    s->state   = STATE_1;
    break;
   case STATE_2:
    /*Preparación mensaje 2*/
    s->textptr = mes2;
    s->textlen = sizeof(mes2);
    break;
  }
}

static void acked(void) {
  struct app2_struct *s = (struct app2_struct *)uip_conn->appstate;

  s->textlen -= uip_conn->len;
  s->textptr += uip_conn->len;
  if(s->textlen == 0) {
    switch(s->state) {
    case STATE_1:
      /*Transición a estado 2*/
      s->state = STATE_2;
      P2OUT ^= BIT1;              // setea P2.1 en cero <=> LED ON
      break;
```

```
    case STATE_2:
      /* ACK2 recibido: cierre de conexión*/
      P2OUT ^= BIT1;              // setea P2.1 en uno <=> LED OFF
      uip_close();
      break;
    }
  }
}

static void senddata(void) {
  struct app2_struct *s = (struct app2_struct *)uip_conn->appstate;

  if(s->textlen > 0) {
    uip_send(s->textptr, s->textlen);
  }
}
```

Es posible verificar el funcionamiento de la aplicación de un modo similar al utilizado en el capítulo anterior. Se utiliza el mismo esquema de conexión y el mismo cliente TCP/IP (fig. 3.3.):

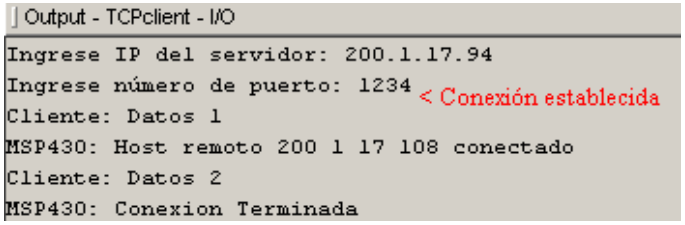


Fig. 3.3. Conexión con cliente TCP/IP

Durante la ejecución, el led de estado confirma la recepción del ACK1 (led ON) y del ACK 2 (led OFF).

3.3. Cliente SMTP para envío de e-mail.

En el punto anterior se programó una aplicación de tipo servidor que actúa según el estado en que se encuentre, de acuerdo al diagrama de la fig. 3.1. No obstante, por el lado del cliente no se tiene un diagrama de estados similar. Es más, el único evento que gatilla la transición entre estados de la aplicación es el envío de datos por parte del cliente, sin importar su tipo, por lo que ni siquiera se le incluyó en el diagrama antes señalado.

Un protocolo estándar de comunicación TCP/IP se caracteriza por el envío de peticiones y respuestas que permiten a las partes (cliente/servidor) constatar que la transferencia de información está siendo llevada a cabo exitosamente. De acuerdo al tipo de petición o respuesta enviada o recibida, las partes configuran su estado: quedan a la espera de nueva información, envían los datos que le fueron requeridos o realizan labores de administración de la conexión, para finalmente terminarla.

Debido a lo anterior, un protocolo se puede modelar utilizando una máquina de estados que involucre a ambas partes. En particular, se tomará el código de una implementación básica del protocolo SMTP que forma parte del stack uIP (`smtp.c`); se utiliza el diseño estructurado de aplicaciones propuesto en el punto anterior para programar un cliente que envíe un correo electrónico al destinatario especificado en el programa cuando se presione uno de los botones de la tarjeta de desarrollo. Este cliente (Easyweb2) cumple con el diagrama de la fig. 3.4. que incluye esta vez como parte fundamental al servidor de correo (mx1.elo.utfsm.cl).

3.3.1. Establecimiento de la conexión y preparación del mensaje.

La primera etapa (1) consiste en establecer la conexión TCP, mediante un 'handshake' de tres vías (tres paquetes iniciales). Esto se realiza cada vez que se desea enviar un correo, mediante la invocación a la función `smtp_send`, que ejecuta el siguiente código:

```
conn = uip_connect(smtpserver, HTONS(25));
  if(conn == NULL) {
    return 0;
  }
```

Como se señaló en el capítulo 2, la función `uip_connect` permite establecer una conexión TCP/IP con el host remoto especificado (`smtpserver`), a través del puerto correspondiente. En este caso, como se trata del protocolo SMTP, se utiliza el puerto 25. Si no es posible abrir la conexión, se sale de la función retornando el valor cero. En caso de éxito, la aplicación pasa al estado inicial `STATE_SEND_NONE` (estado A según fig. 3.4) y se prepara el mensaje a enviar, retornándose un uno:

```
s = (struct smtp_state *)conn->appstate;
  s->state = STATE_SEND_NONE;
  s->sentlen = s->sendptr = s->textlen = 0;
  s->to = to;
  s->from = from;
  s->subject = subject;
  s->msg = msg;
  s->msglen = msglen;
  return 1;
```

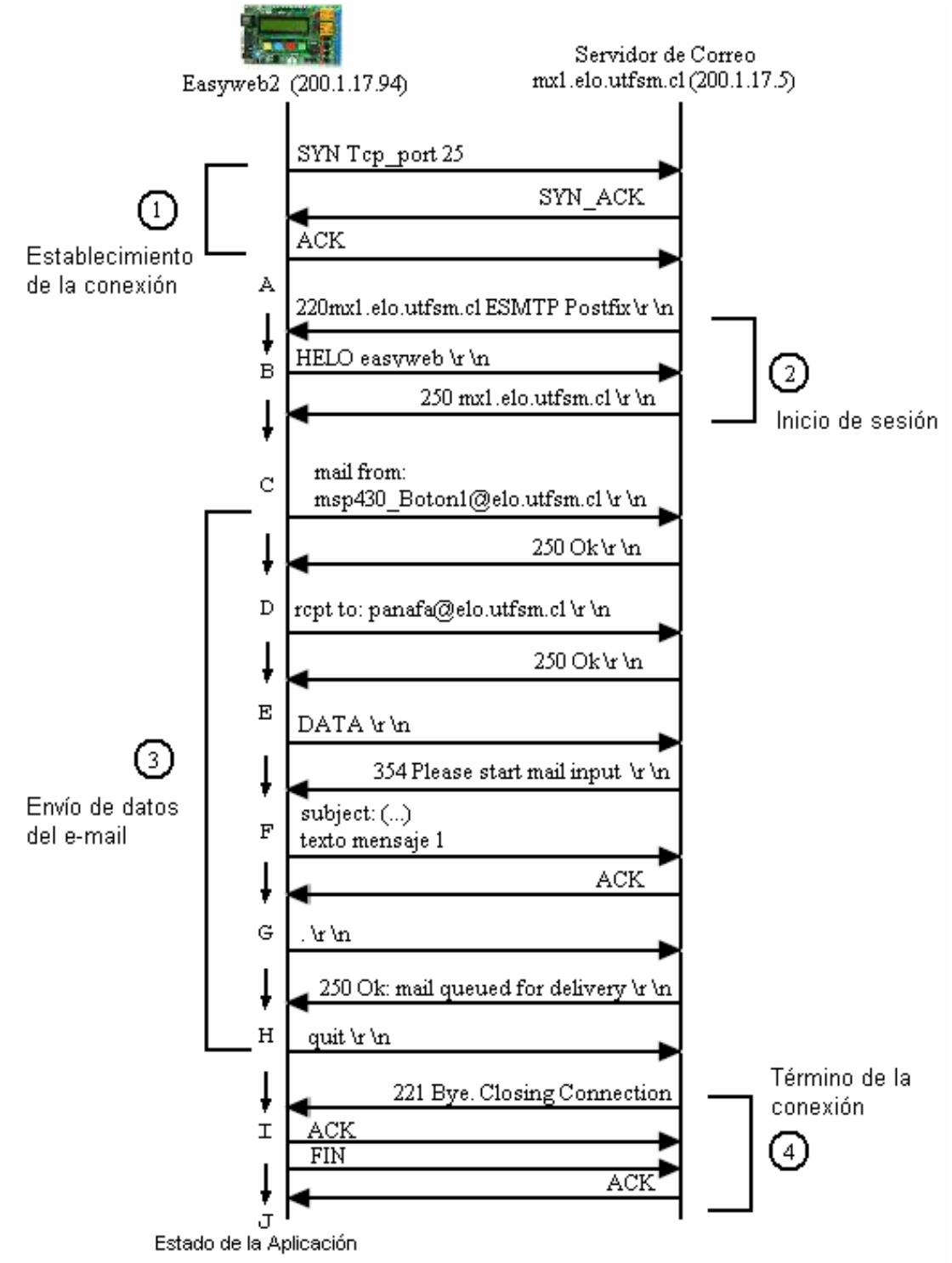


Fig. 3.4. Diagrama del cliente SMTP para envío de correo electrónico.

A la función smtp_send se le deben pasar los siguientes parámetros:

```
unsigned char smtp_send
 (char *to,  char *from,  char *subject, char *msg,  u16_t msglen)
```

to:	dirección del destinatario
from:	dirección del remitente
subject:	asunto del mensaje
msg:	texto del mensaje
msglen:	la longitud en caracteres del texto del mensaje

Es decir, todos ellos corresponden a campos requeridos para preparar el mensaje de acuerdo al protocolo SMTP; para que queden almacenados en RAM el programador debe declararlos de una manera similar a:

```
char *to1 = "panafa@elo.utfsm.cl";
char *from1 = "MSP430_Boton1@elo.utfsm.cl";
char *subject1 = "test 1 easyweb";
char msg1[] = "Texto mensaje 1";
```

Estos string quedan apuntados en la estructura uip_conn gracias al código de preparación de mensaje antes señalado:

```
s->to = to;
s->from = from;
s->subject = subject;
s->msg = msg;
s->msglen = msglen;
```

Donde s es puntero a la estructura de la aplicación smtp.c y tiene una función equivalente a la descrita en el código de la aplicación del punto 3.2.

Para producir el envío del correo (análogamente, la invocación de la función smtp_send) se programaron los eventos presión del botón 1 ó 2 de la tarjeta de desarrollo Easyweb2. El hecho de presionar cada uno de ellos desencadenará el envío de dos correos diferentes (al mismo destinatario, lo que es por comodidad, pero con distinto contenido). De esta manera, el código de manejo de eventos se incluye al final del cuerpo del loop principal de control (fig. 2.2.) y el código resultante es el siguiente:

```
/* Inicializaciones*/
/* Initialize the uIP TCP/IP stack. */
  uip_init();
  uip_arp_init();

  /* Initialize the device driver. */
  cs8900a_init();
```

```
  /* Configure the SMTP module with an SMTP server and the hostname of
the host.*/
  smtp_configure((char *)lhostname, (u16_t *)server);

  InitLCD();
  SEND_CMD(LINEA1);
  printf ("Para enviar mail");
  SEND_CMD(LINEA2);
  printf ("presione B1 o B2");

int
main(void)
{
  char *to1 = "panafa@elo.utfsm.cl";
  char *from1 = "MSP430_Boton1@elo.utfsm.cl";
  char *subject1 = "test 1 easyweb";
  char msg1[] = "Texto mensaje 1";

  char *to2 = "panafa@elo.utfsm.cl";
  char *from2 = "MSP430_Boton2@elo.utfsm.cl";
  char *subject2 = "test 2 easyweb";
  char msg2[] = "Texto mensaje 2";

  while(1) {

      /*loop de control stack uIP*/
        ......
      /*************************/

      //eventos que producen el envio de correo: boton 1 o 2 presionado
      if ((B1)==0) {
        if(smtp_send(to1, from1, subject1, msg1, (u16_t)sizeof(msg1))==0)
        {//No se puede establecer conexion con el servidor SMTP
            SEND_CMD(CLR_DISP);

            printf("   Error21   ");
            Delayx100us(32000);
            SEND_CMD(CLR_DISP);

            printf ("Para enviar mail");
            SEND_CMD(LINEA2);
            printf ("presione B1 o B2");
         }
         else
         {//conexion 1 establecida - mensaje 1 preparado
            SEND_CMD(CLR_DISP);

            printf("Con1 establecida");
            Delayx100us(32000);
            SEND_CMD(CLR_DISP);

            /*printf ("Para enviar mail");
            SEND_CMD(LINEA2);
            printf ("presione B1 o B2");   */
         }
       }
```

```
       if ((B2)==0) {
        if(smtp_send(to2, from2, subject2, msg2, (u16_t)sizeof(msg2))==0)
          {//No se puede establecer conexion con el servidor SMTP
            SEND_CMD(CLR_DISP);

            printf("   Error22   ");
            Delayx100us(32000);
            SEND_CMD(CLR_DISP);

            printf ("Para enviar mail");
            SEND_CMD(LINEA2);
            printf ("presione B1 o B2");
        }
        else
        {// conexion 1 establecida - mensaje2 preparado
            SEND_CMD(CLR_DISP);

            printf("Con2 Establecida");
            Delayx100us(32000);
            SEND_CMD(CLR_DISP);

            /*printf ("Para enviar mail");
            SEND_CMD(LINEA2);
            printf ("presione B1 o B2");*/
         }
        }
      }
```

Es importante señalar que debe declararse la dirección IP del servidor de correo:

```
const char *lhostname="easyweb"; // nombre del host local
#define server_IPADDR0     200U  // direccion IP del servidor de correo
#define server_IPADDR1       1U
#define server_IPADDR2      17U
#define server_IPADDR3       5U


/*arreglo que contiene la dir IP del servidor de correo*/
const u16_t server[2] =
  {  HTONS((server_IPADDR0 << 8) | server_IPADDR1),
     HTONS((server_IPADDR2 << 8) | server_IPADDR3)};
```

Este arreglo es utilizado por la función `smtp_configure` que es invocada en las inicializaciones del código anterior.

A partir del código anterior se desprende que se utiliza el display de LCD de la tarjeta de desarrollo (los `printf` invocan al driver del LCD, cuyo código se encuentra en CD adjunto).

3.3.2. Interfaz de la aplicación.

Al iniciarse el programa, la aplicación despliega una pantalla de bienvenida (fig. 3.5):

Fig. 3.5. Despliegue inicial, cliente SMTP.

Al presionar el botón azul o amarillo, se inicia el envío del correo con la invocación de la función `smtp_send`. Si la conexión se establece exitosamente, se despliega el mensaje (fig. 3.6):

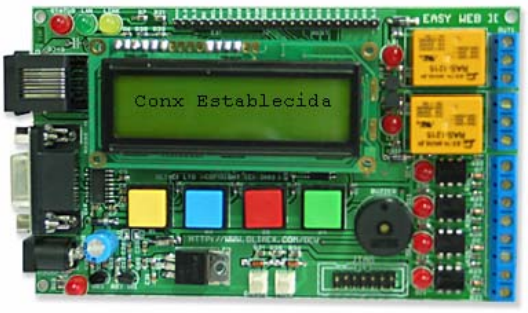


Fig. 3.6. Conexión establecida con el servidor.

En caso contrario, se despliega el mensaje de error: "`Error 2x`", en que x indica el número de la conexión que se intentó establecer (botón 1, amarillo / botón 2, azul).

A continuación se inician las etapas 2, 3 y 4 (fig. 3.4.), de identificación de las partes (2), envío de datos propios del e-mail (3), y cierre de la conexión. Esto es tarea de la aplicación `smtp.c`, con su administrador de eventos, cuyo código se entrega a continuación:

```
void
smtp_appcall(void)
{
  struct smtp_state *s;
  s = (struct smtp_state *)uip_conn->appstate;

  if(uip_connected()) {
    /*    senddata();*/
    return;
  }
  if(uip_acked()) {
```

```
    acked(s);
  }
  if(uip_newdata()) {
    newdata(s);
  }
  if(uip_rexmit() ||
     uip_newdata() ||
     uip_acked()) {
    senddata(s);
  } else if(uip_poll()) {
    senddata(s);
  }

}
```

Como se puede apreciar, se utiliza la misma idea descrita en la aplicación del punto 3.2.

Es por lo tanto el stack uIP, mediante la invocación periódica de la aplicación, el que se encarga de dar inicio a la transición entre los diferentes estados. Éstos son:

```
                                  // Estados según fig. 3.4
#define STATE_SEND_NONE         0  //          A
#define STATE_SEND_HELO         1  //          B
#define STATE_SEND_MAIL_FROM    2  //          C
#define STATE_SEND_RCPT_TO      3  //          D
#define STATE_SEND_DATA         4  //          E
#define STATE_SEND_DATA_HEADERS 5  //          F
#define STATE_SEND_DATA_MESSAGE 6  //          G
#define STATE_SEND_DATA_END     7  //          H
#define STATE_SEND_QUIT         8  //          I
#define STATE_SEND_DONE         9  //          J
```

Como indica el diagrama (fig. 3.4), es la llegada de las respuestas del servidor la que gatilla las transiciones entre los diferentes estados. Estos string recibidos son analizados en la función `newdata` y son utilizados por la aplicación para saber que información enviar. La función que envía los datos, de acuerdo al estado en que se encuentra la aplicación, es `senddata`. A su vez, la función `acked` realiza también algunas transiciones entre estados, e invoca, en el caso que la aplicación se encuentre en el último estado (`STATE_SEND_DONE`), a la función `smtp_done`, la que debe ser programada específicamente para cada aplicación. En este punto, el lector debiera poder comprender, mediante la simple lectura de estas funciones, la acción que ellas realizan.

Debido su longitud, los códigos de las funciones `newdata`, `acked` y `senddata` se entregan en el anexo.

3.3.3. Término del envío y cierre de la conexión.

La función smtp_done, programada para esta aplicación, es la siguiente:

```
void smtp_done(unsigned char error)
{
   if (error == SMTP_ERR_OK)
   //correo enviado
      {
       printf("  Mail Enviado  ");
       Delayx100us(32000);
       SEND_CMD(CLR_DISP);

       printf ("Para enviar mail");
       SEND_CMD(LINEA2);
       printf ("presione B1 o B2");
      }
   else
   //se aborta el envio por comando ilegal.
      {
       printf("   Error1   ");
       SEND_CMD(LINEA2);
       printf ("Reenvie correo");
       Delayx100us(32000);
       SEND_CMD(CLR_DISP);

       printf ("Para enviar mail");
       SEND_CMD(LINEA2);
       printf ("presione B1 o B2");
      }
}
```

Si `smtp_done` es invocada con la macro `SMTP_ERR_OK`, quiere decir que la invocación ha sido efectuada desde la función `acked`, y que el mensaje ha sido enviado exitosamente al servidor de correo. Debido a esto, se despliega en el display el siguiente mensaje (fig. 3.7) :

Fig. 3.7. Envío exitoso del e-mail.

Este mensaje se mantiene por un instante, para luego volver al estado inicial de display de bienvenida (fig. 3.5).

Si ocurre que el parámetro pasado a la función `smtp_done` no es `SMTP_ERROR_OK`, significa que se ha producido algún error en la secuencia de envío de datos, por lo que el servidor ha enviado algún mensaje de error (5xx ...). Es entonces la función `newdata` la que ha invocado a `smtp_done` y se aborta la conexión, desplegándose el mensaje: `"Error 1 / Reenvie correo"`. Luego de un instante, se despliega nuevamente la pantalla de bienvenida.

De este modo, se cuenta con una aplicación que permite el envío de correo electrónico desde un dispositivo pequeño y con recursos de memoria limitados, como es un microcontrolador, más su interfaz de red.

3.3.4. Análisis del tráfico de paquetes.

El análisis de los paquetes IP enviados y recibidos por la aplicación SMTP permite verificar que las etapas propuestas en el esquema de la fig. 3.4 se cumplen a cabalidad. El envío de un correo, a través de la presión de uno de los dos botones, se traduce en la generación de un tráfico de alrededor de 25 paquetes; debido a los retardos incorporados por efectos de lectura del display, el envío del correo mediante esta aplicación toma aproximadamente 5 [s] (fig. 3.8).

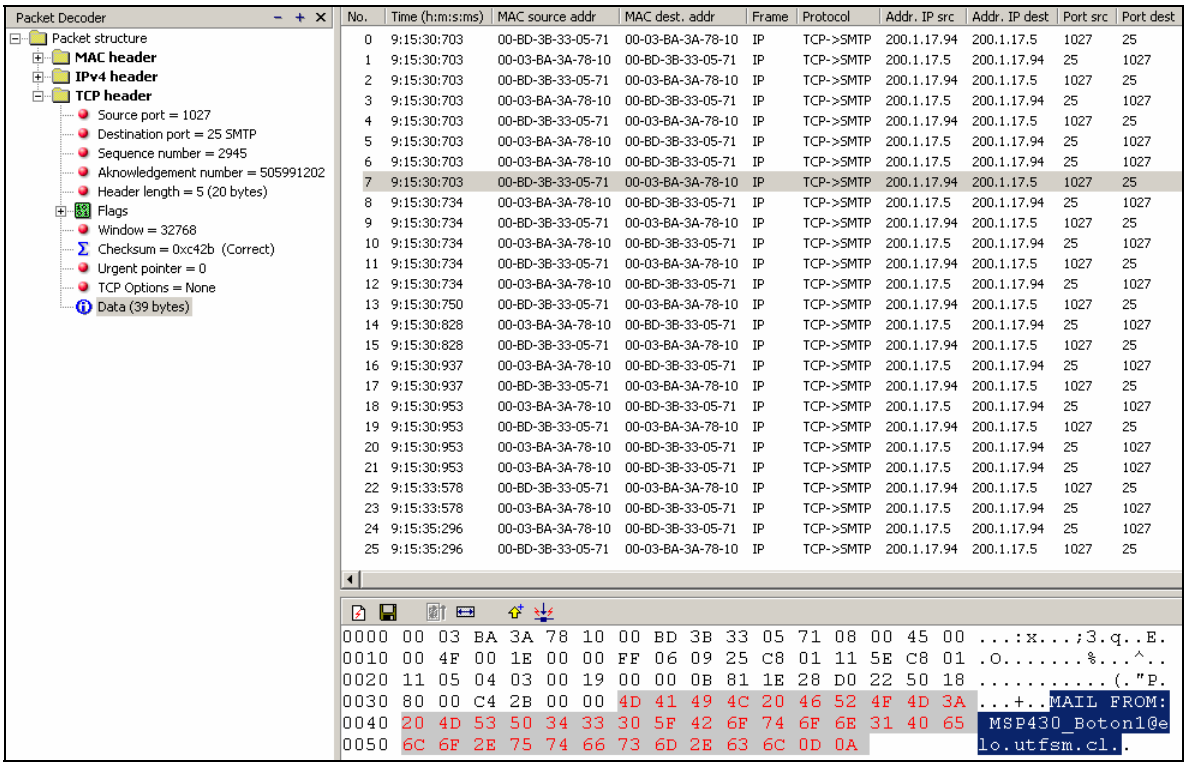


Fig. 3.8. Tráfico de paquetes aplicación cliente SMTP

Claro está que así como se ha propuesto, el envío de un mensaje estático (siempre el mismo) no es una aplicación tan atractiva, a menos que se estuviera pensando en un sistema de alarmas, en el que interese enviar siempre un mismo mensaje por cada evento gatillado en el sistema. Pero la estructura planteada permite la extensión de la máquina de estados (para por ejemplo adjuntar algún

tipo de archivo), y utilizar los mensajes como sistema de información de los datos que está procesando el microcontrolador. Una aplicación de este tipo se realizó en el marco del Seminario de Diseño con Microcontroladores (primer semestre, 2004). Para información al respecto, consultar [26].

Capítulo 4
Control en forma remota de la tarjeta Easyweb2.

Luego de haber experimentado con las opciones que ofrece el stack uIP y de haber programado aplicaciones básicas de red para la tarjeta de desarrollo Easyweb2, es posible iniciar el desarrollo de aplicaciones un poco más elaboradas, pensando en el control y la adquisición de datos en forma remota desde el microcontrolador y su interfaz de red.

En la práctica, la programación de las aplicaciones de red para el microcontrolador no diferirá mucho de lo realizado hasta ahora. Es por el lado de las aplicaciones de usuario, es decir, aplicaciones pensadas para correr en un PC, que se enfocará el trabajo a lo largo de este capítulo.

4.1. Sockets y streams.

Como se señaló en el capítulo 3, en el que se programó un cliente TCP/IP en lenguaje Java, el manejo de funciones de red es sencillo pero a la vez poderoso con Java. La creación de conexiones se realiza mediante sockets, que una vez configurados permiten que en el código restante prácticamente no figuren instrucciones referentes a funciones de red.

Todo el procesamiento de envío y recepción de datos se realiza utilizando los dos streams disponibles en cada socket (uno de entrada y otro de salida, fig. 4.1.), y el encadenamiento de éstos con los llamados streams de procesamiento.

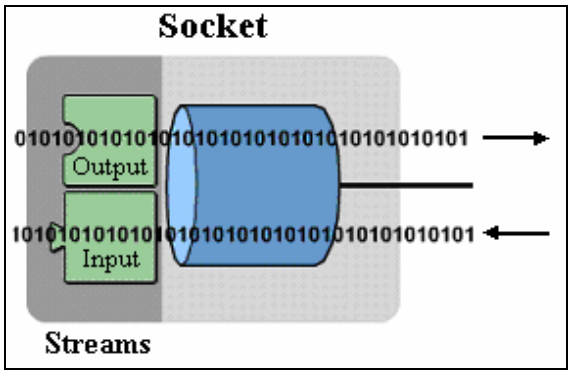


Fig. 4.1. Esquema de un socket y flujo de datos a través de la conexión de red.

Estos últimos permiten transformar de una manera sistemática los datos que están siendo transferidos a través de la conexión establecida mediante sockets (en particular, a través de sus input y output streams). Viene al caso señalar tres de estos '*process/filter streams*', que fueron utilizados en las aplicaciones del capítulo 2:

- `DataInputStream`, `DataOutputStream` : estas clases transforman bytes en primitivas (int, long, double), o vice-versa. Si se requiere leer primitivas desde un stream de entrada, debe encadenarse a éste el stream de

procesamiento `DataInputStream`. Si lo que se desea es escribir primitivas hacia un stream de salida, debe encadenarse a éste un `DataOutputStream`.

- `InputStreamReader, OutputStreamWriter`: estas clases convierten bytes a texto de código del formato standard Unicode, o vice-versa.

- `BufferedReader, BufferedWriter`: estas clases permiten guardar en buffer los datos procedentes de un stream de entrada o que van en ruta hacia un stream de salida. Esto permite mejorar el rendimiento de las aplicaciones, simplificar el manejo de memoria y flexibilizar el tratamiento de los datos.

Es así como la programación de aplicaciones de red se reduce en Java al establecimiento de las conexiones mediante la apertura de sockets y todo el resto tiene que ver con la utilización de los streams apropiados para el envío y recepción de datos.

Estas técnicas de uso de streams se continuarán utilizando a lo largo de este capítulo, en el que el enfoque se traslada a la programación de aplicaciones de usuario que permitan la adquisición de datos y el control en forma remota de la tarjeta de desarrollo Easyweb2.

4.2. Programación de un Cliente TCP/IP con interfaz gráfica de usuario.

El objetivo en este punto es la programación de un cliente TCP/IP en lenguaje Java, por lo que se emplean los conceptos descritos anteriormente en lo referente al establecimiento de las conexiones y al manejo de los flujos de datos, o *streaming*. Pero además de esto, se incorpora el desarrollo de una interfaz gráfica para que el usuario no ejecute acciones desde la línea de comando, sino que a través de una ventana de aplicación diseñada especialmente para el control (encender status-led, abrir-cerrar relés, activar el *buzzer*) y la transferencia de datos en forma remota.

La creación de la aplicación y de la GUI (*Graphic User Interface*) se realizaron con el ambiente de desarrollo para Java Netbeans IDE 3.6, disponible en forma gratuita junto con los *downloads* de Java en el sitio de Sun Microsystems [27].

4.2.1. Creación del JFrame y de los componentes gráficos.

Lo primero a realizar es la creación del JFrame, componente base de la aplicación que sirve para contener a los demás elementos que completarán la GUI. Para esto, debe antes que todo montarse un sistema de archivos en el IDE. Una vez creado el directorio, debe agregarse el Jframe mediante la opción New-> JFrame Form (fig. 4.2.) :

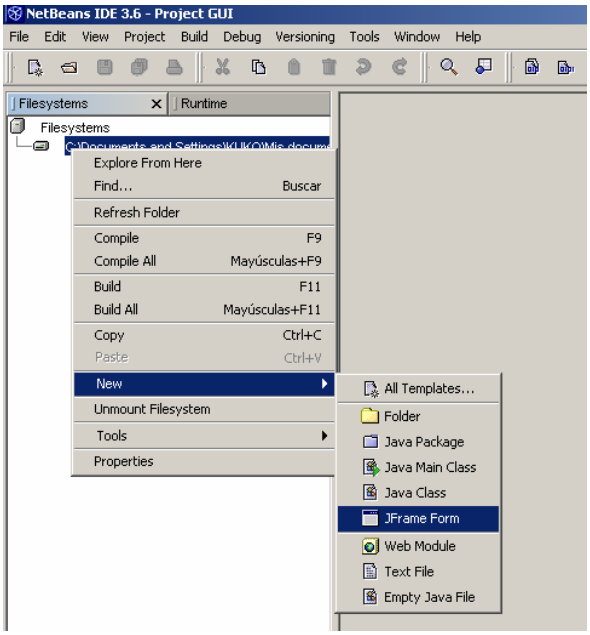


Fig. 4.2. Creación del Jframe

Si se ha efectuado correctamente el paso anterior, incluyendo el paso de ingresar un nombre para el JFrame que se está creando, el IDE debiera mostrar la siguiente configuración (fig. 4.3.) :

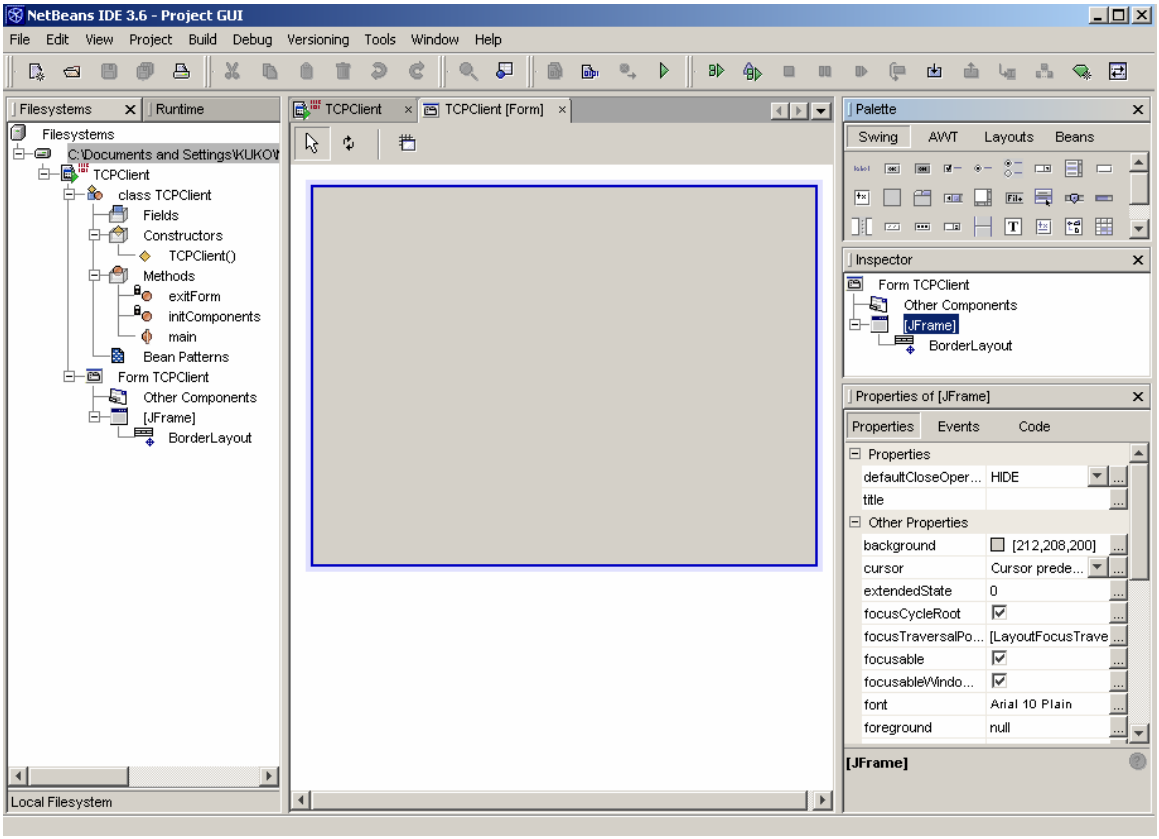


Fig. 4.3. Ambiente de desarrollo de la GUI.

Como se aprecia en la figura anterior, el ambiente de desarrollo queda dividido en tres partes. La ventana de más a la izquierda corresponde al nodo del sistema de archivos y permite acceder a los distintos componentes creados en forma automática y que forman parte de la clase recién creada (en este caso, 'TCPClient'). Éstos son el constructor `TCPClient()` y los métodos `exitForm`, `initComponents` y `main`. El constructor invoca sólo a initComponents, método que permite crear e inicializar las componentes de la GUI. El método `exitForm` permite aguardar por el evento de cerrar la ventana, lo que ocasiona que se salga apropiadamente de la aplicación. Finalmente, el método `main` es creado por omisión con la instanciación e invocación del método `show()`, lo que permite mostrar en pantalla, luego de la compilación (F9) y ejecución (F6), la GUI creada. El código generado en forma automática se muestra en la fig. 4.4. :

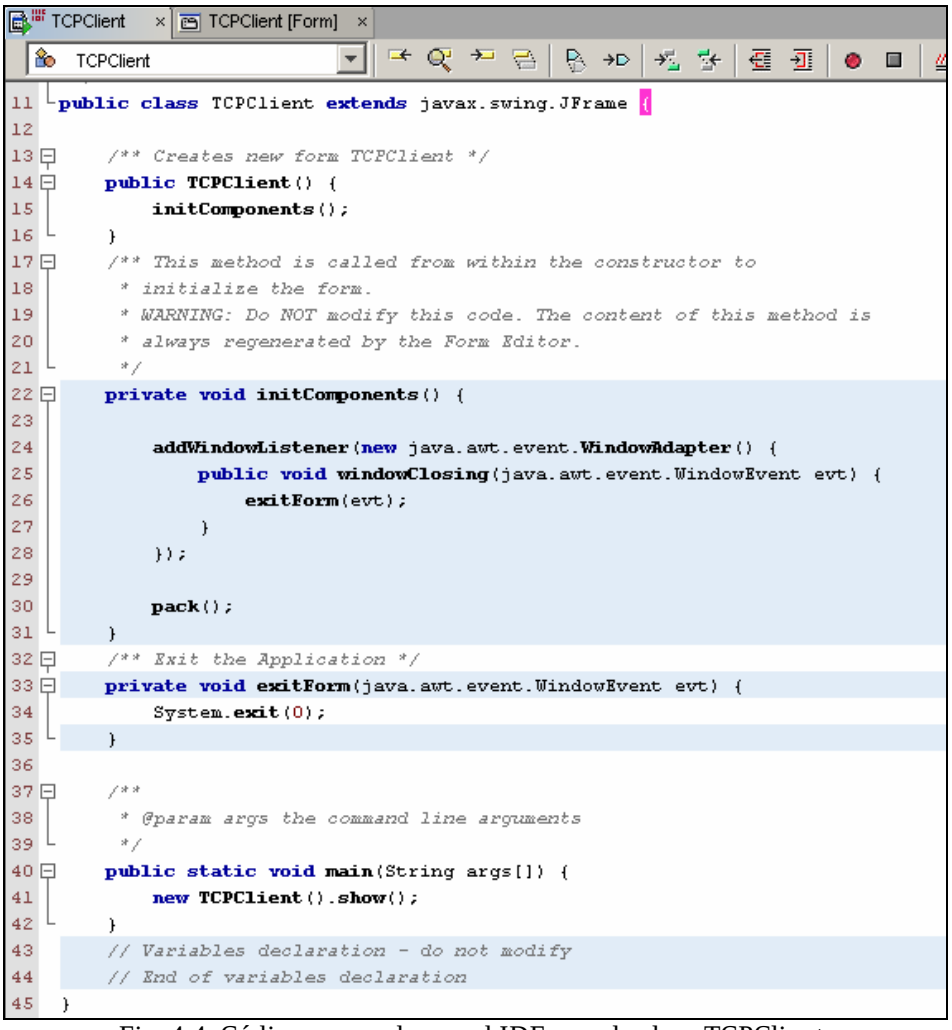


Fig. 4.4. Código generado por el IDE para la clase TCPClient.

Las zonas en azul corresponden a código generado automáticamente por el denominado *Form Editor*. No es posible escribir en estas zonas. A medida que se van añadiendo componentes a la GUI, la zona de código correspondiente a `initComponents` crece. Los aportes del programador a la aplicación deben

incorporarse entonces en los espacios en blanco asignados para este propósito. Por ejemplo, el código para el método `main` es editable, siendo incluso opcional la incorporación de este método en la GUI (por ejemplo, el objeto TCPClient podría ser invocado por otra clase).

En la ventana del medio del ambiente de desarrollo (fig. 4.3.) es posible acceder tanto al código de la clase TCPClient, visto en detalle en la fig. 4.4., como a la representación gráfica de TCPClient (Form). Lo que al inicio es un cuadrado gris vacío, se irá llenando con distintos componentes para el cumplimiento de las funciones de la aplicación.

Finalmente, en las ventanas de más a la derecha del ambiente de desarrollo (fig. 4.3.), se encuentran las componentes de la GUI (*Swing*, *AWT*, *Layouts*, *Beans*), la ventanilla *Inspector*, que permite navegar por las distintas componentes de la GUI y por último la ventanilla *Properties*, que permite editar las propiedades de cada una de las componentes de la GUI.

Es a partir de estas ventanas que se tomarán los botones, checkboxes y campos de texto que formarán parte de la aplicación.

Para insertar un componente en el JFrame, basta seleccionarlo desde la ventana correspondiente y luego hacer click en los límites del JFrame. Por ejemplo, para insertar un botón (jButton) se le selecciona desde los componentes Swing y luego se efectúa un click en el JFrame (fig. 4.5.):

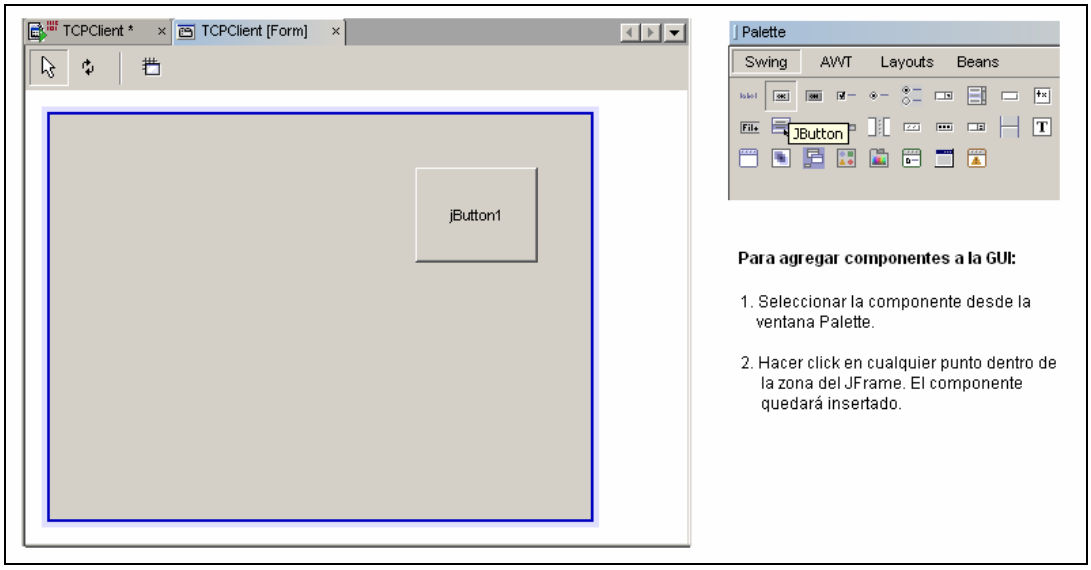


Fig. 4.5. Insertar un componente en el JFrame de la GUI.

De este modo, el componente queda inserto en el marco de la ventana de la aplicación y se genera en forma automática el código correspondiente a la clase TCPClient, en el método `initComponents` (fig. 4.6.):

```
private void initComponents() {
    jButton1 = new javax.swing.JButton();

    getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());

    addWindowListener(new java.awt.event.WindowAdapter() {
        public void windowClosing(java.awt.event.WindowEvent evt) {
            exitForm(evt);
        }
    });

    jButton1.setText("jButton1");
    getContentPane().add(jButton1, new org.netbeans.lib.awtextra.AbsoluteConstraints(270, 40, 90, 70));

    pack();
}
```

Código generado para el componente jButton

Fig. 4.6. Código generado para el componente jButton.

Es posible notar que se genera una línea extra de código:

```
getContentPane().setLayout(new org.netbeans.lib.awtextra.AbsoluteLayout());
```

Esto corresponde a la configuración del *Layout* de la ventana de aplicación, es decir, a la disposición de las componentes en el JFrame. En este caso, la configuración del JFrame como *Absolute Layout* permite al programador ubicar, según su propio diseño, las componentes de la GUI.

De este modo, se insertan todas las demás componentes que formarán parte de la aplicación: 4 botones, 4 campos de entrada de texto, 3 checkboxes y un área de texto (fig. 4.7.) :

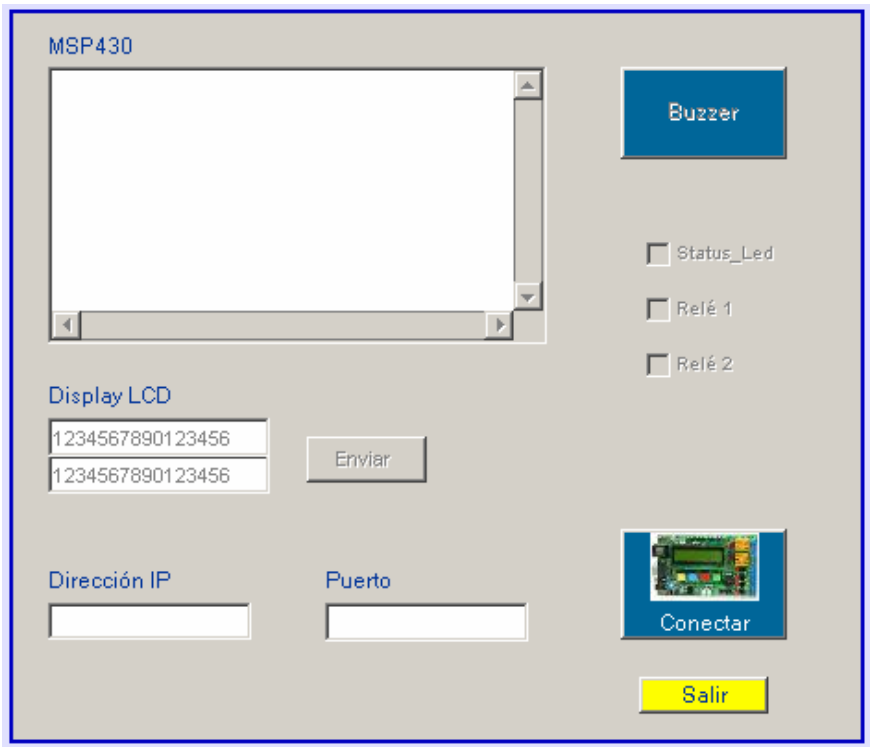


Fig. 4.7. Componentes de la GUI.

4.2.2. Programación de los eventos para las componentes de la GUI.

Esta aplicación gráfica permitirá ingresar la dirección IP y el puerto a través de los cuales se efectuará la conexión TCP con la tarjeta de desarrollo Easyweb2, en la que correrá un servidor programado específicamente para esta aplicación. Al presionar en el botón 'Conectar', se establecerá la conexión con la interfaz de red del microcontrolador MSPF149. Si la conexión se efectúa exitosamente, el servidor enviará el mensaje: "Conexión establecida", que será capturado por el cliente y desplegado en el área de texto con rótulo MSP430.

A partir de ese minuto, se podrán activar los checkboxes (para manejar el led de estado y los relés), presionar en el botón 'Buzzer' para la activación del zumbador presente en la tarjeta y se podrán enviar mensajes al display, todo esto en forma remota.

Estas acciones deben ser programadas en función de los eventos correspondientes. Por ejemplo, debe programarse que al presionar en el botón 'Conectar' se recojan los datos de dirección IP y puerto desde los campos de entrada de texto correspondientes y con esta información abrir un socket para establecer la conexión TCP. Esto se realiza mediante la ventana *Properties* correspondiente al Botón 'Conectar'. Debe seleccionarse el campo *Events*, y seleccionar el evento `mouseClicked`. Esto permite que el programador inserte el código requerido en la zona apropiada del método `jButton1MouseClicked` (fig. 4.8.):

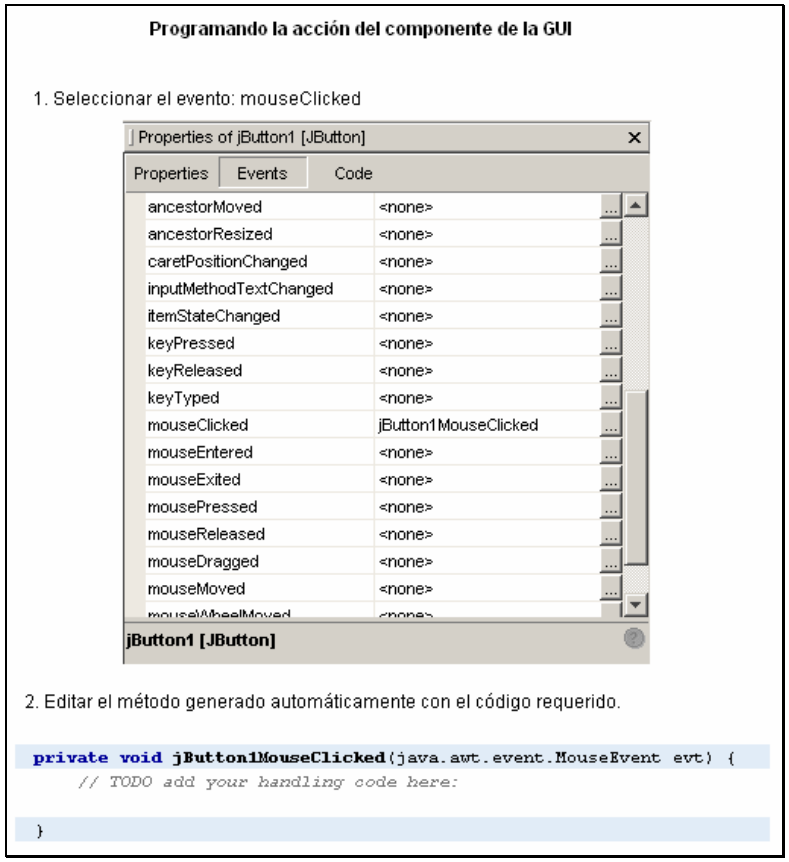


Fig. 4.8. Programación de eventos para un componente de la GUI.

Para que se cumpla la acción antes descrita (establecer la conexión), el método correspondiente al botón 'Conectar' se programa según:

```
private void jButton1MouseClicked(java.awt.event.MouseEvent evt) {
   // toma el texto de la dir IP
   String serverHost = jTextField1.getText();

   // toma el string del puerto, lo castea a integer
   int serverPort = Integer.parseInt(jTextField2.getText());

   try {
     //abre el socket
     clientSocket = new Socket(serverHost, serverPort);

     // inicializacion de los streams
     outToServer = newDataOutputStream(clientSocket.getOutputStream());
     inFromServer=
new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));

     // inicio del polling
     ok = true;

     //conexion establecida, por lo que activa los restantes componentes
     jCheckBox1.setEnabled(true);
     jCheckBox2.setEnabled(true);
     jCheckBox3.setEnabled(true);
     jButton3.setEnabled(true);
     textArea1.setEnabled(true);
     jTextField4.setEnabled(true);
     jTextField5.setEnabled(true);
     jButton4.setEnabled(true);
   }
   catch( IOException e ) {
     System.out.println( e );
   }
}
```

Esta es la parte más importante, ya que el resto consistirá sólo en el envío de datos a través de los streams creados (`outToServer, inFromServer`). Una vez que se establece la conexión, el servidor responde con el string "`Conexion Establecida`", línea que es capturada a través del método `poll()` :

```
public void poll() throws IOException {

        if (ok) {
            // bloquea la ejecucion hasta leer linea finalizada en '\n'
            String uIPmsg = inFromServer.readLine();

            // escribe el string en el area de texto
            textArea1.setText(uIPmsg);
        }
    }
```

Cabe señalar que se eliminó el método `main`, creado por omisión por el IDE. Se creó otra clase, TCPClientTest, para crear una instancia del objeto TCPClient e invocar al método poll periódicamente y así recibir cualquier flujo de datos que termine con un '\n'. El código de TCPClientTest es el siguiente:

```
public class TCPClientTest {

    public static void main(String[] args) throws IOException {

        TCPClient cliente = new TCPClient();
        cliente.show();
        for (;;) {
            cliente.poll();
        }
    }
}
```

Una vez recibido y desplegado en la zona de texto de la aplicación el mensaje de establecimiento de la conexión, enviado por el servidor del MSP430, se está en condiciones de presionar los restantes botones y checkboxes y de enviar el texto que se desea desplegar en el LCD de la tarjeta de desarrollo Easyweb2. Esto se logra gracias a los códigos presentados en la fig. 4.9.

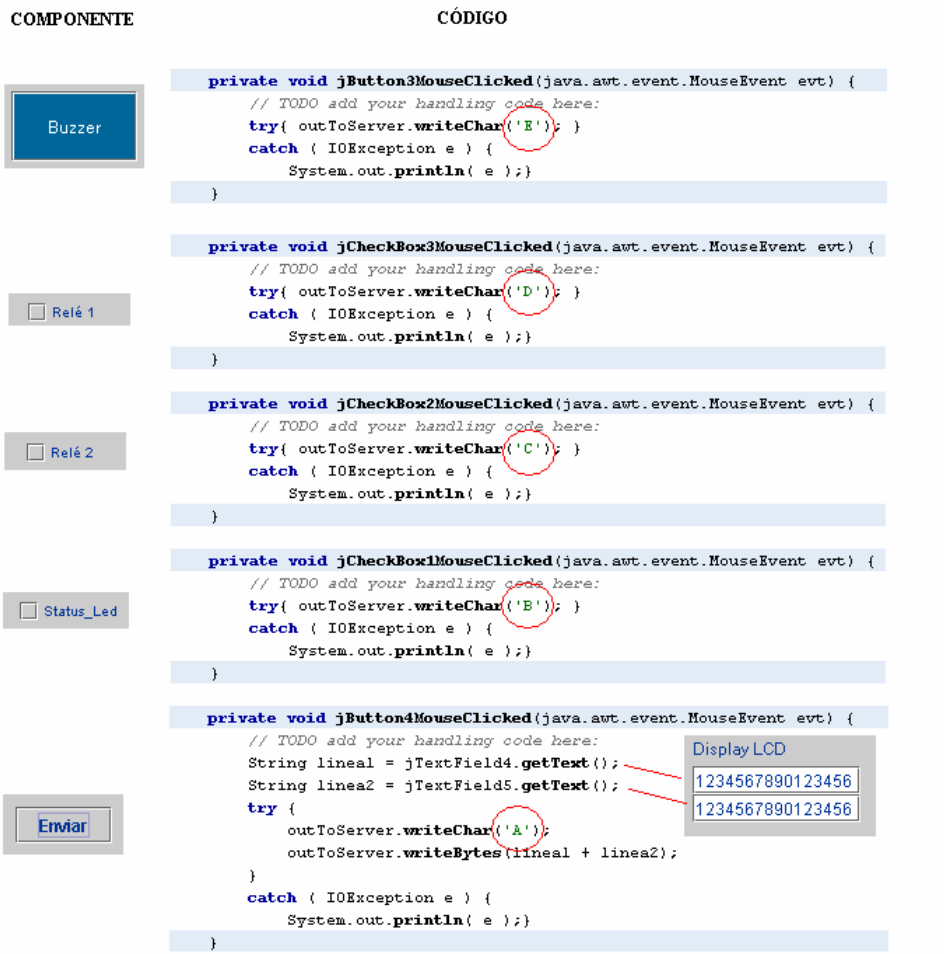


Fig. 4.9. Código de atención de eventos por componente de la GUI.

De un modo análogo al descrito en la figura 4.9., el presionar el botón 'Salir' significa el envío del caracter de control 'F', el cierre de la conexión TCP y el término de la aplicación; el código es:

```
private void jButton2MouseClicked(java.awt.event.MouseEvent evt) {
        try {
            clientSocket.close();          // cierra el socket
            outToServer.writeChar('F');
        }
        catch (IOException e) {
            System.out.println(e);
        }
        System.exit(0);                    // termino aplicacion
    }
```

Para un tutorial paso a paso acerca del desarrollo con Netbeans IDE 3.6 de aplicaciones con interfaz gráfica para el usuario, dirigirse a [28].

4.2.3. Programación del servidor.

Una vez que el cliente está debidamente programado, debe desarrollarse la aplicación del lado del servidor, que correrá en la tarjeta de desarrollo Easyweb2. Esta aplicación debe cumplir las siguientes tareas:

- Escuchar por conexiones a través de un determinado puerto.
- Al establecer una conexión con un cliente remoto, enviar el mensaje "`Conexion Establecida`", para efectos de verificación por parte del cliente.
- Interpretar los datos recibidos en `uip_appdata` (buffer de datos del stack).
- Ejecutar la acción que corresponda.

La siguiente tabla resume la acción ejecutada ante la llegada de un determinado caracter:

Caracter Recibido	Acción realizada
'A'	Despliegue del string en LCD
'B'	Conmutación del Status-Led
'C'	Apertura/Cierre del Relé 1
'D'	Apertura/Cierre del Relé 2
'E'	Activación del Buzzer
'F'	Término de la conexión

Esta aplicación contará con tres estados: no conectado, conectado y salir:

```
#define NO_CONECTADO 0
#define CONECTADO    1
#define SALIR        2
```

Con lo anterior claramente especificado, sólo resta programar la función de inicialización de la aplicación y las funciones invocadas por el administrador de eventos; los códigos son, respectivamente:

```
void appserver_init(void) {
   struct app3_struct *s = (struct app3_struct *)uip_conn->appstate;

   InitPorts();               // configuracion de puertas digitales
   uip_listen(HTONS(1510));   // escucha a traves del puerto 1510
   InitLCD();                 // inicializa el LCD

   s->state = NO_CONECTADO;   // configura estado inicial
}
```

```
void call_appserver(void) {
  struct app3_struct *s;
  s = (struct app3_struct *)uip_conn->appstate;

  if(uip_aborted()) {
    aborted();
  }
  if(uip_timedout()) {
    timedout();
  }
  if(uip_closed()) {
    closed();
  }
  if(uip_connected()) {
    connected(s);
    senddata();
  }
  if(uip_acked()) {
    acked(s);
  }
  if(uip_newdata()) {
    newdata(s);
    senddata(s);
  }
}
```

Las funciones `aborted`, `timedout` y `closed` configuran el estado de la conexión en NO_CONECTADO:

```
static void aborted(struct app3_struct *s) {
  s->state = NO_CONECTADO;
}
static void timedout(struct app3_struct *s) {
  s->state = NO_CONECTADO;
}
static void closed(struct app3_struct *s) {
  s->state = NO_CONECTADO;
}
```

La función connected permite cambiar el estado de la aplicación a CONECTADO y efectúa el envío del mensaje de establecimiento de la conexión:

```
static void connected (struct app3_struct *s) {
  s->state = CONECTADO;
  s->textptr = welcome;
  s->textlen = sizeof(welcome);
}
```

Es la función newdata la que cumple el papel más relevante, ya que es la encargada de identificar el comando de control recibido y actuar según lo especificado previamente:

```
#define LCD         'A'
#define STATUS_LED  'B'
#define RELE1       'C'
#define RELE2       'D'
#define BUZZER      'E'
#define FIN         'F'

static void newdata (struct app3_struct *s) {
  // recepcion de comandos para activación de periféricos
  int ct = 0;
  int i = 0;

  switch (*(char *)uip_appdata) {

   case LCD:
    SEND_CMD(CLR_DISP);
    SEND_CMD(LINEA1);
    for (i=1; i < 33; i++) {
     SEND_CHAR((*(char *)(uip_appdata + i)));
     if ( i == 16) SEND_CMD(LINEA2);
    }
    break;

   case STATUS_LED:
    P2OUT ^= BIT1;
    break;

   case RELE1:
    P1OUT ^= BIT5;
    break;

   case RELE2:
    P1OUT ^= BIT6;
    break;

   case BUZZER:
      while (ct < 1000) {
      SEND_CMD(CLR_DISP);
        BUZ1_OFF;
      BUZ2_ON;
      Delay(_100us);        //buzzer 5 000 Hz
```

```
      BUZ2_OFF;
      BUZ1_ON;
      Delay(_100us);
        ct ++;
      }
    break;

   case FIN:
    s->textptr = exit;
    s->textlen = sizeof(exit);
    s->state = SALIR;
    break;

   default: break;
 }
}
```

Por último, las funciones `senddata` y `acked` efectúan el envío y la recepción de datos, respectivamente. La función `acked` cumple en este caso sólo con la verificación del correcto envío de los datos y con el cierre de la conexión, si se ha llegado al estado SALIR. Los códigos son:

```
static void acked(struct app3_struct *s) {
  s->textlen -= uip_conn->len;
  s->textptr += uip_conn->len;
  if (s->textlen == 0) {
   switch (s->state) {
    case CONECTADO:
     break;
    case SALIR:
     uip_close();
     break;
    default:
     break;
   }
  }
}

static void senddata(struct app3_struct *s) {
  if(s->textlen > 0) {
    uip_send(s->textptr, s->textlen);
  }
}
```

De este modo, se cuenta con el par de aplicaciones cliente/servidor que permiten el control en forma remota de la tarjeta Easyweb2. En la figura 4.10. se entrega un ejemplo de la ejecución de las aplicaciones:

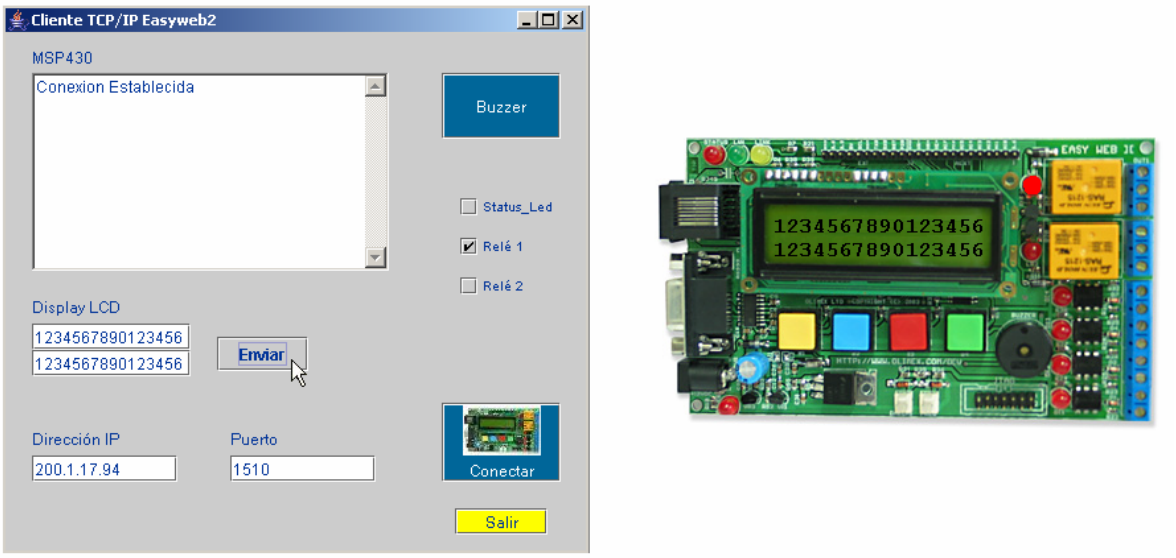


Fig. 4.10. Aplicaciones cliente GUI – servidor Easyweb2.

Los códigos completos de ambas aplicaciones se encuentran en el CD adjunto.

4.3. Desarrollo de aplicaciones para su ejecución en un servidor web.

Al momento de desarrollar una aplicación, el programador debe considerar como uno de los factores importantes el conjunto de requerimientos de hardware y software para la posterior ejecución de la aplicación. Dentro de ciertos márgenes, la idea es desarrollar aplicaciones que sean independientes de la plataforma utilizada y cuyos requerimientos adicionales de software sean los mínimos.

Cuando un usuario se encuentra con el código de una aplicación como la presentada en el punto anterior, tiene que verificar una serie de requerimientos antes de poder ejecutarla, incluso si este código ya se encuentra compilado. Una posibilidad es incrustar la aplicación en una página web en la forma de un *applet* de Java. Aún así, el browser debe tener instalados los plug-ins necesarios para la correcta visualización de las componentes gráficas.

El programador del stack uIP consideró la alternativa de acceder a la tarjeta de desarrollo Easyweb2 a través de un browser. Además del cliente SMTP (cuya implementación se describe en el capítulo 3, punto 3), de un cliente web y de un servidor Telnet, junto con el stack se incluye también el código para la implementación de un servidor web en la tarjeta de desarrollo. Es decir, es posible programar páginas web, convertirlas junto con los demás archivos que la componen a formato hexadecimal y dejarlas en la memoria del microcontrolador para su posterior acceso a través de un browser. Esto implicó la programación de un sistema de archivos (`fs.c`), funciones de script CGI (`cgi.c`), y el código correspondiente al servidor web propiamente tal (`httpd.c`) [14].

Lo anterior constituye un gran aporte al desarrollo de aplicaciones de red para la conectividad de microcontroladores. Los códigos son abiertos y se encuentran funcionales. Sin embargo, además de la complejidad involucrada en el diseño de esta aplicación, hay que considerar varios aspectos:

- Las aplicaciones deben ser portadas: a menos que se cuente con el compilador CrossStudio de la Rowley [22], el programador debe portar los códigos al compilador de su preferencia, lo que no necesariamente resulta sencillo. Debe tenerse especial precaución con el tratamiento de los datos almacenados en memoria.

- Los recursos de memoria son un factor delicado en el diseño con microcontroladores: la programación de un servidor web en un microcontrolador llama de inmediato la atención, pero siempre existirá un límite en cuanto a la versatilidad del contenido de las páginas web servidas, dada la escasa capacidad de almacenamiento de datos.

- Poca flexibilidad en el mantenimiento de la aplicación: si el programador necesita actualizar las páginas web, debe re-diseñarlas, volverlas a convertir al formato hexadecimal, actualizar los datos del sistema de archivos y descargar nuevamente la aplicación a la memoria del microcontrolador. Este procedimiento es, en su conjunto, objetivamente complicado.

- Función de un microcontrolador: no se trata de ser inflexible ni menos aún desconocer el avance en el diseño con microcontroladores. Por el contrario, mientras más alternativas existan, mejor. Pero no hay que olvidar la función esencial de los microntroladores, que es la adquisición de datos y el control de procesos. Mientras más recursos estén destinados a esto y más livianas sean las aplicaciones que el dispositivo ejecute, mejor será el rendimiento del sistema embebido.

Teniendo en cuenta todas estas observaciones, se ha pensado en la necesidad de plantear un enfoque distinto a la hora de querer cumplir con el objetivo de acceder a la tarjeta de desarrollo Easyweb2 en forma remota y a través de un browser.

La idea central es liberar los recursos del microcontrolador asociados a la programación de un servidor web como aquel planteado en [14]. Para lograr lo anterior, se ha pensado en diseñar aplicaciones que corran en un servidor web y que permitan tanto la conexión TCP/IP con la tarjeta de desarrollo, para la transferencia de comandos de control y de datos "crudos", como el tratamiento de todos los aspectos involucrados con la presentación de la página web asociada a la aplicación.

4.3.1 Servlets.

La manera más tradicional de invocar programas desde un servidor web es el mecanismo denominado *Common Gateway Interface* (CGI), que surgió desde los inicios del desarrollo del HTML. Desde el browser del cliente, y en forma de un formulario HTML, se envían datos hacia el programa CGI en el servidor. La URL ingresada determina qué programa CGI se ejecutará (puede ser un script o un archivo ejecutable). Los datos son procesados y se genera una respuesta, normalmente una página HTML. Esta respuesta es devuelta al servidor web, que empaqueta esta página web y la envía en forma de una respuesta HTTP, la que es finalmente recibida en el browser del cliente.

Por el conjunto de características relacionadas con la programación de aplicaciones de *networking*, ya descritas en los capítulos 2 y 3, se continuará utilizando el lenguaje Java para el desarrollo de las aplicaciones que serán contenidas en el servidor web. En particular, este tipo de aplicaciones se denominan **servlets**.

Sun Microsystems desarrolló los servlets como un avance respecto a la tecnología tradicional de CGI [29]. Un servlet Java es un programa que, al igual que su similar en CGI, corre en un servidor web. El tipo de tareas que uno y otro pueden realizar son similares. Los servlets también procesan peticiones HTTP, ejecutan acciones (consultas a bases de datos, o conexiones a un host remoto, que es lo que interesa aquí) y generan respuestas en forma dinámica, por ejemplo una página web. No obstante, las arquitecturas de ejecución de una y otra tecnología son distintas.

A diferencia de los programas CGI, los servlets se ejecutan dentro de un contenedor web. Éste es en la práctica una máquina virtual Java que entrega una implementación de la API de Java para servlets (paquete `javax.servlet`). De esta manera, un servlet es una instancia de un HTTPServlet (podrían emplearse también los denominados GenericServlet), administrada por el contenedor web para responder ante peticiones HTTP, mediante los métodos `doPost()` o `doGet()`.

Las principales ventajas de los Java servlets son las siguientes:

- Cada petición HTTP tiene su propio hilo de ejecución, por lo que el procesamiento es más rápido que con un programa CGI. Estos últimos manejan cada petición como un proceso aparte del sistema operativo.

- Los servlets están orientados a objetos. En particular, el manejo de funciones de red con Java (las mismas ya utilizadas en los capítulos anteriores) es considerablemente más sencillo que con Perl o cualquier otro lenguaje que se utilice para programar los scripts CGI.

- El contenedor web entrega servicios adicionales a los servlets, como manejo de errores y seguridad.

En la figura 4.11 se entrega un esquema de la arquitectura de ejecución de un servlet ante una petición HTTP. Se incluye además la opción de efectuar a través del servlet una conexión TCP con la tarjeta de desarrollo Easyweb2, que es el principal objetivo a implementar.

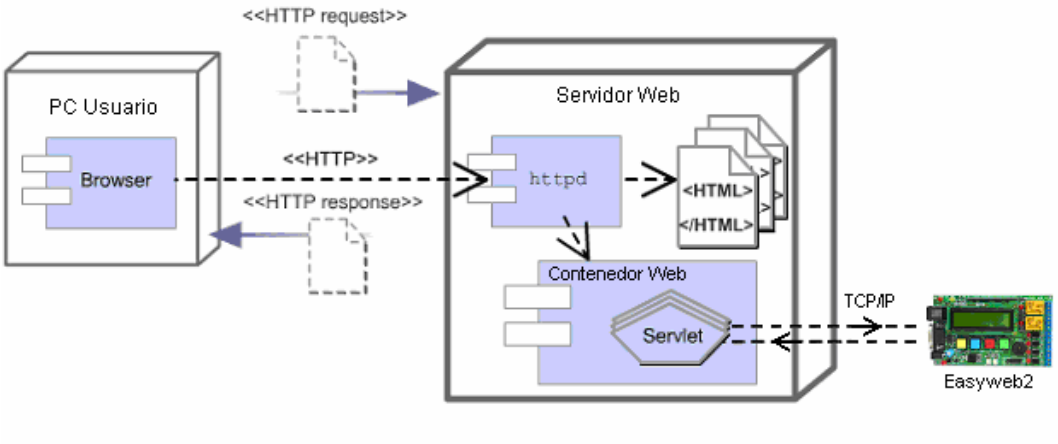


Fig. 4.11. Esquema de la aplicación web a implementar.

Esta arquitectura permitirá que cualquier usuario, sin más requerimientos que un browser, acceda mediante una URL determinada a una página web, complete los campos del formulario HTML ofrecido y los envíe hacia el servidor web; el servlet alojado será el encargado de establecer una conexión TCP con la tarjeta de desarrollo Easyweb2 e intercambiará datos que posteriormente serán devueltos y presentados en la forma de una página HTML.

4.3.2. Programación de una aplicación web: HTML y servlet.

Al igual que todas las aplicaciones desarrolladas en Java a lo largo de este trabajo, para programar la aplicación web se utlizó el Netbeans IDE 3.6. Este ambiente de desarrollo incorpora un servidor Tomcat 5 para el despliegue y la ejecución de los módulos web creados. Para iniciar o detener este servidor Tomcat, hay que seleccionar desde la ventana de *Runtime* el nodo Tomcat 5 Server, desplegar con click derecho el menú de opciones y hacer click en Start/Stop Server. Posteriormente, es posible verificar el correcto funcionamiento del servidor mediante un browser ingresando la URL http://localhost:8084 (fig. 4.12.).

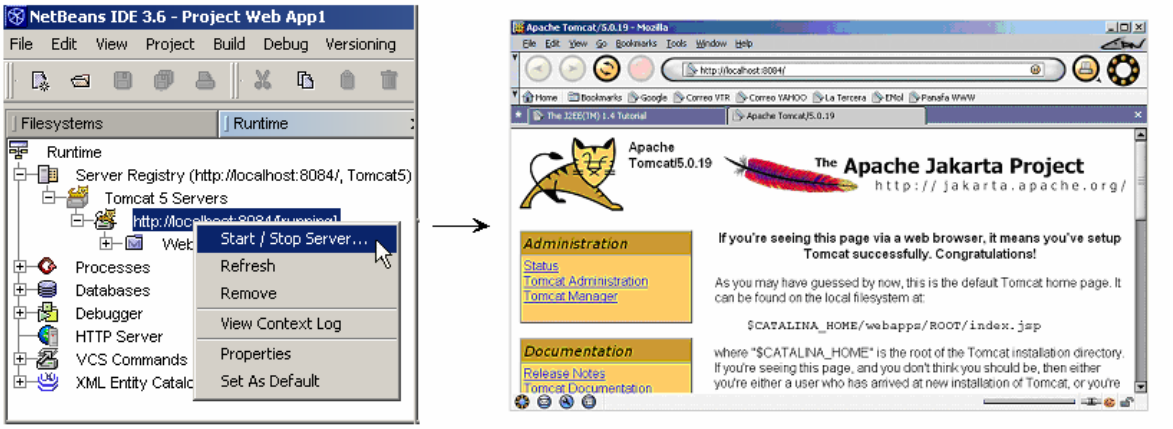


Fig. 4.12. Iniciando el servidor web incorporado en Netbeans IDE 3.6.

Para la creación de una aplicación web, debe montarse en el sistema de archivos el directorio en el cual se pretende guardar la página HTML y el servlet. Una vez montado, se selecciona la opción New -> Web Module. Luego de ingresar el nombre correspondiente, se verifica la siguiente estructura de nodos (fig. 4.13):

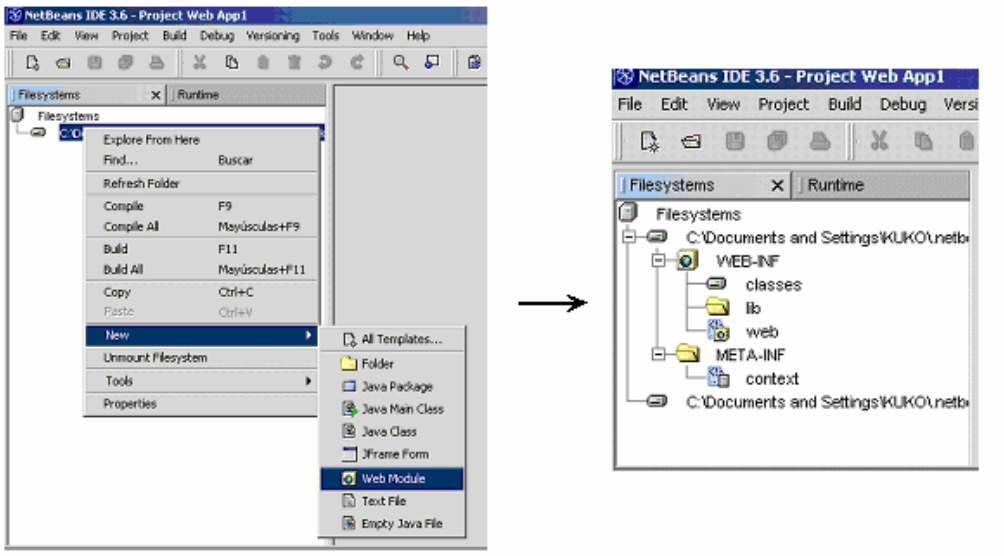


Fig. 4.13. Creación de la estructura de directorios de la aplicación web.

Dada esta estructura, es conveniente indicar en forma precisa en que directorios poner los archivos que componen la aplicación web (fig. 4.14):

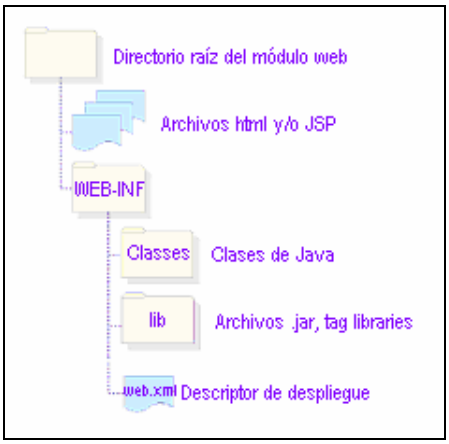


Fig. 4.14. Ubicación de los archivos en la estructura de directorios del módulo web.

Como se propone en la fig. 4.11., esta aplicación consistirá en una página HTML y un servlet; la primera permitirá ingresar los datos (dirección IP y puerto) que se envían en un formulario HTML para que el servlet establezca una conexión TCP con la tarjeta de desarrollo. Finalmente, éste devolverá una página web que incluye los datos recibidos (contenido dinámico).

4.3.2.1. Diseño y despliegue de la página web.

Al margen del texto y del diseño gráfico de la página web, lo que interesa aquí es el código HTML que corresponde a los componentes del formulario:

```
<form action="/WebApp1/EasywebServer" method="post">

        <%tipo de letra%>
        <p><font face="Courier New, Courier, mono">Direcci&oacute;n IP</font>

        <%textos de entrada al formulario%>
        <input name="iphost" type="text" id="iphost" value="200.1.17.94">
        <font face="Courier New, Courier, mono"> Puerto</font>
        <input name="puerto" type="text" id="puerto" value="1510">
        </p>

       <%botón de envío del formulario%>
        <input type="submit" name="Submit" value="Conectar">
</form>
```

La primera línea indica el método de envío del formulario (POST) y la acción realizada: el servlet al cual está dirigido el formulario es el indicado por la ruta encerrada por las comillas.

Los componentes del formulario son dos campos de entrada de texto y un botón de envío del formulario. Los valores de los campos de texto podrán ser accedidos por el servlet a través de sus respectivos nombres (`“iphost”, “puerto”`). Cabe señalar que se han programado por omisión los valores configurados en la tarjeta Easyweb2 (`IP: 200.1.17.94; puerto: 1510`).

Una vez que ha sido diseñada y desplegada en el servidor web (presionar desde el IDE F6 sobre el archivo easyweb.html), la página se accede a través de la URL indicada en la fig. 4.15 :

Fig. 4.15. Página web para el ingreso de los datos de conexión.

4.3.2.2. Programación del servlet.

Para la programación del servlet, se crea el directorio 'Easywebserver' dentro del nodo 'Classes' y se elige la opción New->Servlet. A través de un menú, debe escogerse un nombre para la clase y un patrón de mapeo de la URL. Este paso es importante ya que permite hacer coincidir la acción del formulario HTML con la configuración del servidor web, efectuada a través del descriptor de despliegue `web.xml`. Para este caso, la descripción del servlet queda (extracto del archivo `web.xml` correspondiente a esta aplicación):

```
<servlet>
    <servlet-name>easyServlet</servlet-name>
    <servlet-class>Easywebserver.easyServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>easyServlet</servlet-name>
    <url-pattern>/Easywebserver</url-pattern>
</servlet-mapping>
```

Lo anterior se justifica dado que se eligió como nombre de la clase `easyServlet.java` y se escogió como mapeo de URL la ruta `/Easywebserver` (tomado desde la raíz de la aplicación web, en este caso `WebApp1`).

Al momento de la creación de la clase, el IDE crea varios métodos en forma automática. Tanto estos últimos, como la estructura de la aplicación web, se aprecian en la figura 4.16.

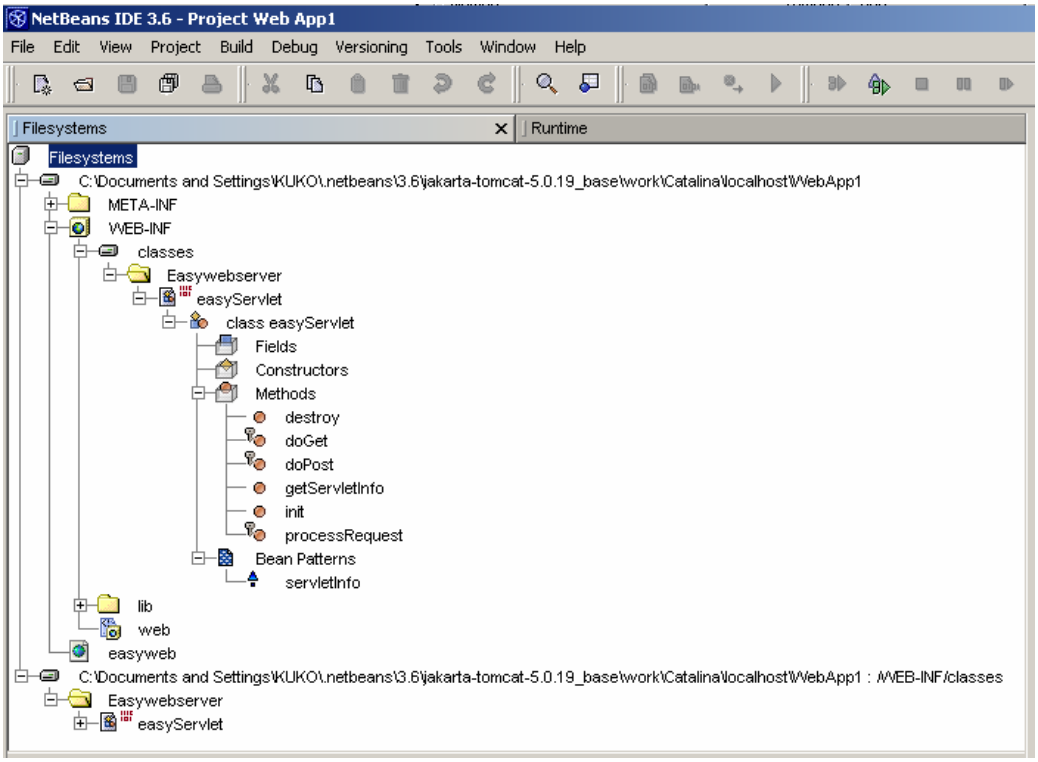


Fig. 4.16. Estructura de directorios de la aplicación web y métodos por omisión del servlet recién creado.

Los métodos `init` y `destroy` corresponden al ciclo de vida del servlet, administrado por el contenedor web. No es necesario editarlos. El método `getServletInfo` es opcional.

Dada la presencia de los métodos `doGet` y `doPost`, es claro que el servlet creado hereda sus métodos y atributos de la clase HttpServlet. Bastaría entonces con programar el cuerpo de ambos métodos para responder a requerimientos HTTP del tipo GET o POST. No obstante, en la práctica ambos métodos invocan a un tercero, `processRequest`, lo que permite programar el procesamiento de la respuesta sin importar a través de qué método fue enviada la petición (todo este código es generado automáticamente por el IDE, fig. 4.17.).

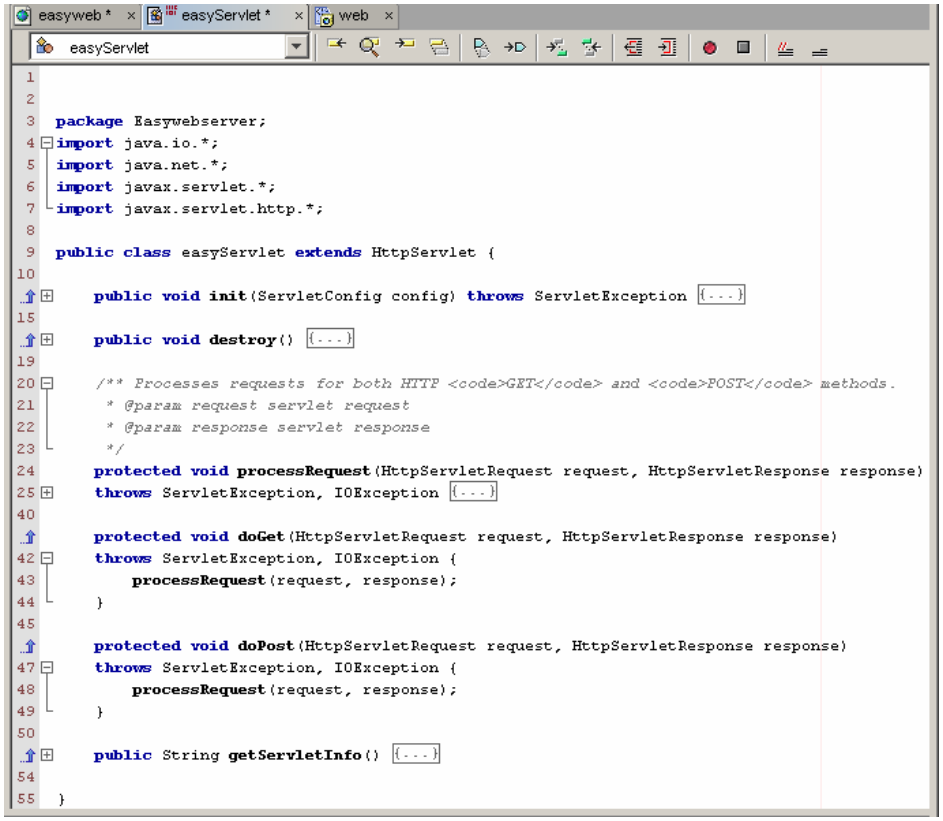


Fig. 4.17. Código de la clase recién creada easyServlet.java

Notar que los parámetros de los métodos `doGet`, `doPost`, y por lo tanto de `processRequest`, corresponden a objetos del tipo `HttpServletRequest` y `HttpServletResponse`. Éstos no son más que la encapsulación a objeto de los streams de las peticiones y respuestas HTTP, efectuada por el contenedor web.

El código por omisión para el método processRequest es:

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        /* TODO output your page here
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet</title>");
        out.println("</head>");
        out.println("<body>");

        out.println("</body>");
        out.println("</html>");
         */
        out.close();
    }
```

Es decir, configura el tipo de contenido de la respuesta y muestra el método para enviar la respuesta en forma de una página HTML (a través del objeto `out`).

Para cumplir con el objetivo de la implementación, se programaron dos métodos extra y se declararon cinco atributos, además de completar el método `processRequest` con las instrucciones adecuadas. Los cinco atributos son:

```
//socket y streams
private Socket mspSocket;
private DataOutputStream outToServer;
private BufferedReader inFromServer;

//strings para la recepcion de datos desde tarjeta Easyweb2
private String uipMsg1 = "null1";
private String uipMsg2 = "null2";
```

Mediante el método `connect`, se establece la conexión TCP con la tarjeta de desarrollo Easyweb2 y se captura el mensaje de confirmación de la conexión:

```
public void connect(String ip, int port) {
   try {
     //establece conexion
     mspSocket = new Socket(ip, port);

     //inicializacion streams
     outToServer =  new DataOutputStream(mspSocket.getOutputStream());
     inFromServer = new BufferedReader(new
                    InputStreamReader(mspSocket.getInputStream()));

     //bloquea ejecucion hasta recibir linea terminada en '\n'
     uipMsg1 = inFromServer.readLine();
   }
   catch (IOException e) {
     System.out.println( e );
   }
}
```

El método `Rele1ON` realiza el envío del caracter de control 'R' hacia la tarjeta Easyweb2, para la conmutación del Rele 1, y captura el segundo mensaje que verifica la recepción del caracter de control:

```
public void Rele1ON_OFF() {
   try {
      outToServer.writeChar('R');
      uipMsg2 = inFromServer.readLine();
   }
   catch (IOException e) {
      System.out.println( e );
   }
}
```

Finalmente, el código para el método processRequest es:

```
protected void processRequest(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //se almacena en ip el valor del componente del formulario de
        //nombre iphost, gracias al metodo getParameter
        String ip = request.getParameter("iphost");

        // id. para el componente de nombre puerto
        String port = request.getParameter("puerto");
        int i = Integer.parseInt(port);

        //se efectua la conexion
        connect(ip, i);

        //se envia el comando de control
        Rele1ON_OFF();

        /* Página de respuesta*/
        out.println("<html>");
        out.println("<head>");
        out.println("<title>MSP430 Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("La ip ingresada es: " + ip + "<BR>");
        out.println("El puerto ingresado es: " + port + "<BR>");
        out.println("Estado de la Conexión?  <BR><BR>”)
        out.println("MSP430 responde: <BR><BR>");
        out.println("<p>");
        out.println("<font size='4' face='Courier New, Courier, mono'>");

        //insercion de contenido dinamico: uipMsg1 y 2
        out.println("<strong>" + uipMsg1 + "</strong><BR><BR>");
        out.println("<strong>" + uipMsg2 + "</strong>");
        out.println("</font></p>");
        out.println("</body>");
        out.println("</html>");

        out.close();
}
```

4.3.2.3. Programación de la aplicación servidor Easyweb2.

Para la comunicación con la aplicación web descrita en el punto anterior, debe programarse una aplicación del tipo servidor en la tarjeta Easyweb2 que escuche por conexiones a través del puerto 1510. Una vez establecida la conexión, enviará el mensaje: "Conexion Establecida." y pasará del estado NO_CONECTADO a CONECTADO. Si luego de esto se recibe el caracter de control 'R', la aplicación enviará el mensaje: "RELE1 ON/OFF, Conexion terminada.", y pasará al estado salir. Luego de recibir el ACK del último envío, terminará la conexión.

Se trata de una aplicación bastante más sencilla, pero conceptualmente equivalente a aquella descrita en el punto 5.2.3., por lo que no se entrará en mayores detalles. El código correspondiente es:

```
#include <string.h>
#include <msp430x14x.h>
#include "uip.h"
#include "appweb1.h"

#define RELE1       'R'

#define NO_CONECTADO 0
#define CONECTADO    1
#define SALIR        2

unsigned char welcome[25] = "1. Conexion Establecida.\n";
unsigned char rele[38] = "2. RELE 1 ON/OFF, Conexion terminada.\n";

//inicializacion de la aplicacion
void appweb1_init(void) {
   struct appweb1_struct *s = (struct appweb1_struct *)uip_conn->appstate;

   P1SEL = 0;
   P1OUT = 0;
   P1DIR = BIT5;                    //habilitar P1.5 (rele1)
   uip_listen(HTONS(1510));         //escucha por conexiones
   s->state = NO_CONECTADO;
}
//administrador de eventos
void call_appweb1(void) {

  struct appweb1_struct *s;
  s = (struct appweb1_struct *)uip_conn->appstate;

  if(uip_aborted()) {
    aborted();
  }
  if(uip_timedout()) {
    timedout();
  }
  if(uip_closed()) {
    closed();
  }
  if(uip_connected()) {
    connected(s);
    senddata();
  }
  if(uip_acked()) {
```

```
    acked(s);
  }
  if(uip_newdata()) {
    newdata(s);
    senddata(s);
  }
}
static void aborted(struct appweb1_struct *s) {
  s->state = NO_CONECTADO;
}
static void timedout(struct appweb1_struct *s) {
  s->state = NO_CONECTADO;
}
static void closed(struct appweb1_struct *s) {
  s->state = NO_CONECTADO;
}

//ACK recibido
static void acked(struct appweb1_struct *s) {
  s->textlen -= uip_conn->len;
  s->textptr += uip_conn->len;
  if (s->textlen == 0) {
   switch (s->state) {
    case CONECTADO:
     break;
    case SALIR:
     uip_close();
     break;
    default:
     break;
   }
  }
}

//conexion establecida
static void connected (struct appweb1_struct *s) {
  s->state = CONECTADO;
  s->textptr = welcome;
  s->textlen = sizeof(welcome);
}

// recepción de comando para activación de rele 1
static void newdata (struct appweb1_struct *s) {
  switch (*(char *)uip_appdata) {

   case RELE1:
    P1OUT ^= BIT5;
    s->state = SALIR;
    s->textptr = rele;
    s->textlen = sizeof(rele);
    break;
   default:
    break;
 }
}
//envio de datos
static void senddata(struct appweb1_struct *s) {
  if(s->textlen > 0) {
    uip_send(s->textptr, s->textlen);
  }
}
```

4.3.2.4. Ejecución de la aplicación web.

Para ejecutar la aplicación web, se debe primero desplegar desde el IDE el módulo web completo hacia el servidor. Para esto, hay que situarse sobre el directorio WEB-INF y presionar F6. Posteriormente, se debe acceder a la página `easyweb.html` del modo indicado en la fig. 4.15 . Finalmente, basta con hacer click en aceptar para iniciar el envío de los parámetros del formulario hacia el servlet. Éste realiza la conexión, recibe el primer mensaje enviado por la tarjeta Easyweb2, envía el comando de control para la conmutación del relé, recibe el segundo mensaje enviado desde la tarjeta y, por último, prepara la página de respuesta que incluye ambos mensajes , a continuación de "MSP responde:" (fig. 4.18).

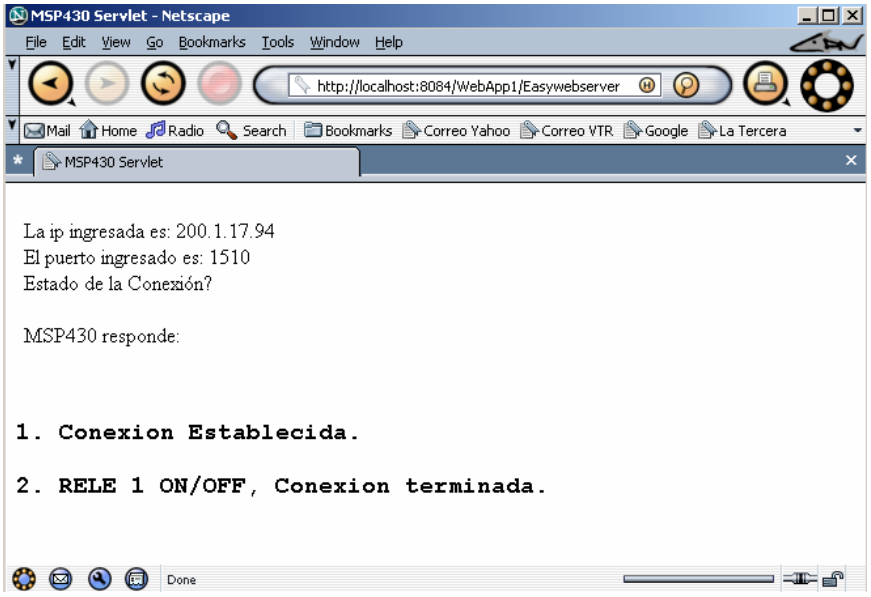


Fig. 4.18. Página de respuesta enviada por el servlet.

4.3.3. Aplicaciones web mediante plantillas (*templates*).

La principal desventaja que presenta el uso exclusivo de servlets para el desarrollo de aplicaciones web es la nula separación entre el diseño de la presentación gráfica de la aplicación (página HTML de respuesta) y la programación de funciones que tienen que ver con el procesamiento y utilización de los datos adquiridos a través del requerimiento HTTP efectuado. Por ejemplo, en el caso de la aplicación anterior, el establecimiento de una conexión y la posterior administración de la transferencia de datos se trataron en el mismo método (`processRequest()`) en el que se abordó la generación de la página de respuesta.

Mezclar la lógica de presentación de una aplicación web con la lógica 'funcional' de ella (*presentation v/s business logic*) implica que cada vez que una página web cambie, los servlets involucrados deben ser reprogramados, recompilados y vueltos a desplegar en el servidor. Esto llevó al desarrollo de las llamadas *template pages*, o plantillas, entre las que se encuentra la tecnología de las JavaServer Pages (JSP).

4.3.3.1. Páginas JSP [30].

La tecnología de Sun JSP es una de las maneras de implementar el concepto de páginas HTML que incluyan incrustaciones de código con una función distinta a la sola presentación de la aplicación web. Existen otras dos tecnologías para la creación de páginas HTML con código embebido: PHP de Apache y ASP de Microsoft. La desventaja de estas últimas es que trabajan exclusivamente con su propio tipo de servidores web.

La estructura de estas plantillas es fundamentalmente la misma: una página HTML que incluye delimitadores (*tags*) para el código que debe ser ejecutado por el servidor web en caso de efectuarse un requerimiento asociado. De esta manera, las páginas JSP son el opuesto de los servlets. Mientras éstos son código Java que contiene HTML, las páginas JSP son HTML con código Java incrustado en ellas. Las páginas JSP son convertidas por el servidor web en servlets que procesan los requerimientos HTML recibidos por la aplicación (fig. 4.19). Por esto, JSP incluye toda la funcionalidad potencial de un servlet.

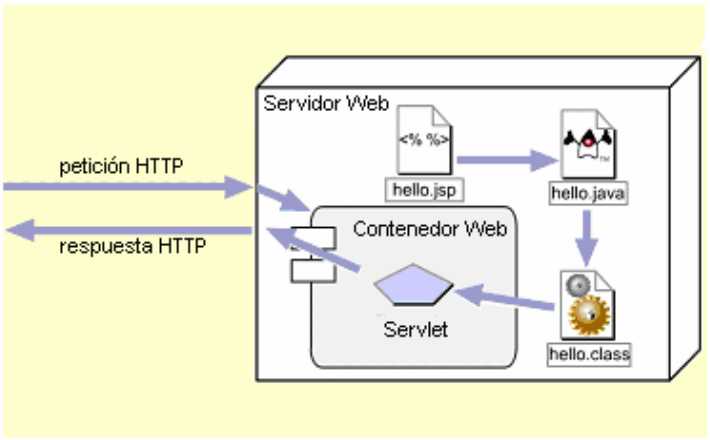


Fig. 4.19. Procesamiento de una página JSP en el servidor.

4.3.3.2. Componentes JavaBeans.

Los componentes JavaBeans son clases de Java que pueden ser fácilmente reutilizadas para la creación y/o modificación de aplicaciones web. Cualquier clase de Java que cumpla con las siguientes dos convenciones es un componente JavaBeans: en primer lugar, debe tener propiedades definidas con métodos `get` y `set` para acceder al valor de éstas o modificarlo. Por ejemplo, para la propiedad de escritura y lectura `puerto`, de tipo entero, deben existir los métodos `getPuerto` y `setPuerto`. En segundo lugar, deben tener un constructor que no reciba argumentos [31].

La tecnología JSP soporta el uso directo de componentes JavaBeans mediante el uso del delimitador estándar `<jsp: use bean…>`. No se ahondará más en este aspecto puesto que el trabajo de desarrollo de una aplicación web más compleja que la del punto anterior, con separación completa entre presentación y función, se realizará mediante el uso del *framework* JSF (JavaServer Faces). No obstante, cabe señalar que un Bean tiene un marco de acción bien delimitado dentro del cual es visible, el que es definido al momento de su creación (fig. 4.20):

Fig. 4.20. Visibilidad (*scope*) de un Bean dentro de la aplicación web.

4.3.3.3. JavaServer Faces [32].

JavaServer Faces (JSF) es un marco de desarrollo en lenguaje Java para la construcción de interfaces de usuario web. Del mismo modo en que Swing entrega la posibilidad de crear aplicaciones con una interfaz gráfica de usuario (punto 5.2.), JSF pone a disposición del programador diversas componentes (botones, hipervínculos, checkboxes) para el desarrollo de interfaces de usuario web y métodos para procesar los eventos generados por el cliente, como por ejemplo un click en uno de los botones que componen la aplicación [31].

Mediante el uso de un IDE apropiado, JSF permite el rápido desarrollo de aplicaciones que:

- Asocian componentes de la interfaz de usuario web a datos del lado del servidor.

- Permiten la reutilización y extensión de funcionalidad de los componentes de la interfaz de usuario.

- Guardan el estado de la interfaz de usuario una vez finalizada la sesión.

La principal ventaja de la tecnología JSF es que ofrece una separación definitiva entre la lógica de presentación web y el código de aplicación propiamente tal. Las aplicaciones web programadas sólo con JSP realizan esta separación sólo en parte. Además, una aplicación JSP no permite mapear peticiones HTTP directamente a un administrador de eventos específico para una componente de la interfaz de usuario, cosa que se logra en forma relativamente sencilla con JSF.

Finalmente, aunque JSF incluye una biblioteca de tags para representar componentes de interfaz de usuario en una página JSP, permite el uso de otra tecnología de presentación, diferente a JSP, además de la creación de nuevos componentes para la interfaz de usuario.

4.3.3.4. Selección de IDE y programación de aplicación web final.

El IDE escogido para el desarrollo de la aplicación web final, utilizando el framework JSF, es el Java Studio Creator v.2004 Q2 [33]. La elección radica en que está basado en NetBeans, por lo que la migración resulta muy natural para el programador acostumbrado a ese IDE. Además, se trata de un ambiente de desarrollo especialmente concebido para la creación de aplicaciones web que utilicen componentes JSF, de tal modo que su uso es muy similar al descrito previamente para la aplicación con componentes GUI de Swing (punto 5.2.). Actualmente, a pesar que JSF es una tecnología emergente, se encuentran disponibles otros ambientes de desarrollo que incorporan este framework. Para mayor información, consultar [34].

En la práctica, la aplicación final a programar no diferirá mucho en cuanto a su funcionalidad de aquella descrita en el punto 5.2. Lo que marca una diferencia radical es que esta aplicación permitirá el control de la tarjeta Easyweb2 (encendido de led, manejo de relés, envío de mensajes a display, activación del buzzer) y la adquisición de datos (recepción de mensajes y monitoreo de la entrada de uno de los canales del conversor análogo-digital) a través de una interfaz web, es decir, a través de un browser. Esto permite cumplir con los objetivos descritos en el punto 5.3. , en cuanto a la menor complejidad en el desarrollo de la aplicación, mayor flexibilidad en la mantención de las páginas web y mayor cercanía de la aplicación con respecto al usuario común; éste tan sólo deberá levantar un browser, ingresar una URL y así tendrá acceso al control de procesos y a la adquisición de datos a través de un sistema embebido, todo en forma remota.

Con la excepción del bloque correspondiente al servidor web, el esquema que resume esta aplicación final es el mismo que el de la figura 4.11. Bastaría

reemplazar este bloque por su similar de la figura 4.19 para que quede descrito en su totalidad el funcionamiento de la aplicación, cuyos detalles de programación se entregan en el siguiente punto.

4.3.3.5. Páginas JSP y reglas de navegación.

Aunque no se trata de un requisito inprescindible, ya que con una sola página podría haber bastado para la aplicación completa, se incluyó una página de bienvenida. Esto permite la escalabilidad de la aplicación (podría utilizarse la página de bienvenida para otros fines además de los aquí descritos) y la menor complejidad en la programación de las páginas.

En resumen, la página de bienvenida `Page1.jsp` permite ingresar la dirección IP y el número de puerto en el que está escuchando la aplicación que corre en la tarjeta de desarrollo Easyweb2. El link 'Conectar' permite entregar estos datos al `SessionBean1.java` (fig. 4.22), para el establecimiento de la conexión, y es además un enlace a la segunda página `panelcontrol.jsp`, que es la que contiene los componentes para el control y la adquisición de datos desde la tarjeta. Tiene además el enlace 'Atrás' para volver a la página de bienvenida. El IDE permite tener una visión gráfica de estas páginas y crear las reglas de navegación de una forma simple: uniendo la componente en cuestión con la página deseada desde la ventana PageNavigation (fig. 4.21):

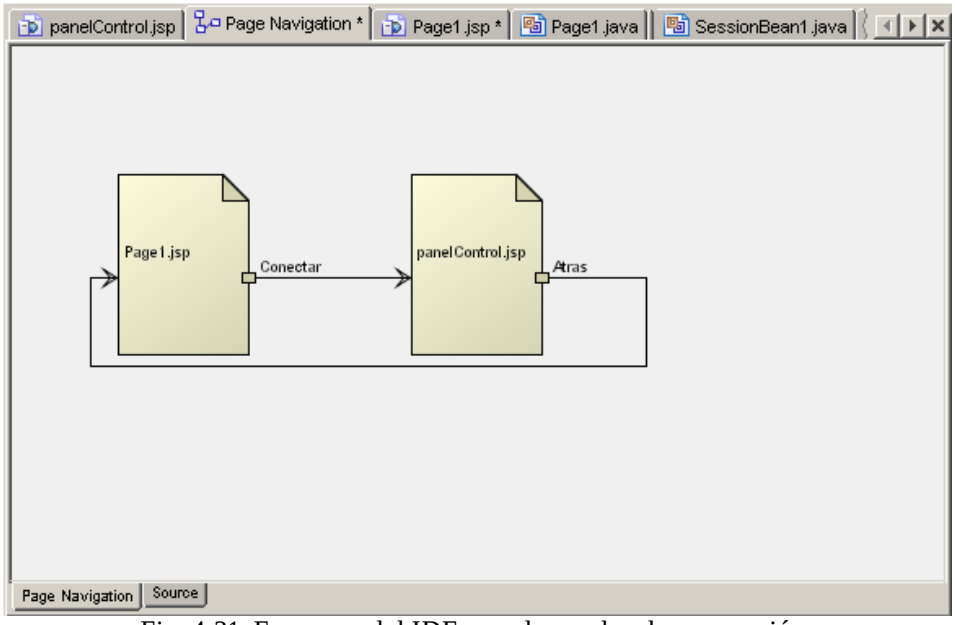


Fig. 4.21. Esquema del IDE para las reglas de navegación.

Al hacer click en cualquiera de las páginas, éstas se amplían y se puede observar el conjunto de componentes que forman parte de ellas. Las componentes se pueden conectar mediante flechas a las otras páginas para la creación automática de las reglas de navegación. Si se hace click en la viñeta 'Source', del

extremo inferior izquierdo, se tiene acceso al código del archivo `faces-config.xml`, que es el que contiene estas reglas. Por ejemplo, para este caso el código creado es:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE faces-config PUBLIC "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config
1.0//EN" "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">
<faces-config>
    <navigation-rule>
        <from-view-id>/panelControl.jsp</from-view-id>
        <navigation-case>
            <from-outcome>Atras</from-outcome>
            <to-view-id>/Page1.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>

    <navigation-rule>
        <from-view-id>/Page1.jsp</from-view-id>
        <navigation-case>
            <from-outcome>Conectar</from-outcome>
            <to-view-id>/panelControl.jsp</to-view-id>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

4.3.3.6. Estructura de directorios del proyecto.

Tanto la página de bienvenida como el Bean con *scope* de sesión tienen los nombres por omisión que entrega el IDE al momento de crear una nueva aplicación web (`Page1.jsp` y `SessionBean1.java`). También se encuentran creados por omisión los Beans de página y de aplicación, aunque este último no se utilizará en esta aplicación. En el descriptor xml `ManagedBeans` se configura el alcance de los distintos Beans utilizados, y en general no es necesario editarlo.

Al crear una nueva página JSP (en este caso `panelControl.jsp`) se crea automáticamente el Bean de página asociado. Todo esto se resume en la ventana de navegación del proyecto (fig. 4.22.):

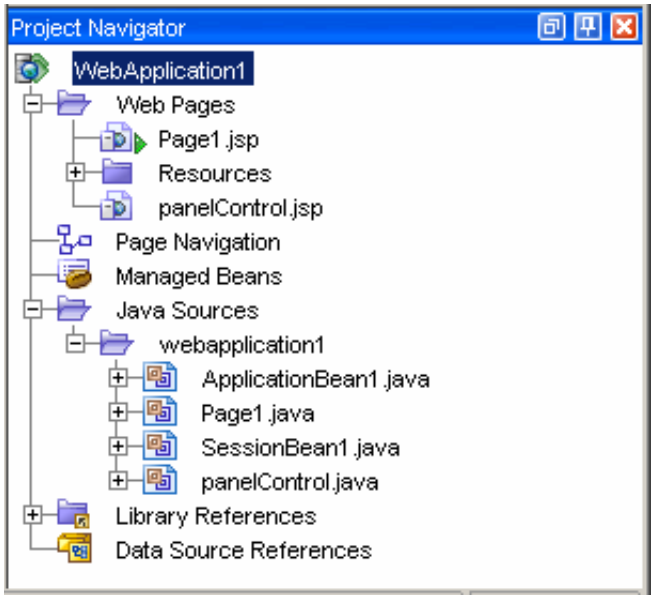


Fig. 4.22. Ventana de navegación del proyecto.

Una de las ventajas de este IDE es que no es necesario preocuparse en tiempo de desarrollo de la estructura de directorios de la aplicación web. Basta seguir el esquema propuesto y es posible desplegar sin problemas la aplicación web al servidor de aplicaciones, incluído con la instalación del programa. Si se necesita agregar más páginas, simplemente se las crea en el directorio `Web Pages`. Todos los recursos asociados a estas páginas (imágenes, por ejemplo) quedan en la carpeta `Resources`. Este esquema de trabajo acelera de sobremanera la programación de una aplicación web.

4.3.3.7. Preparación de las páginas y programación de los Beans.

La página de bienvenida contiene 3 componentes JSF: dos cuadros de entrada de texto y un hipervínculo (fig. 4.23). Éstos quedan representados en el código HTML como *tags* especiales en medio de los formularios de la página. Todo el resto, que tiene que ver con el texto incluído y el diseño gráfico de la página, se realiza directamente en HTML en el cuerpo de la página JSP.

Fig. 4.23. Página de bienvenida.

El código en HTML con los tags especiales para las componentes JSF es el siguiente:

```
<h:form binding="#{Page1.form1}" id="form1">
      <h:inputText binding="#{Page1.textField1}" id="textField1" maxlength="15"
      style="height: 25px; left: 168px; top: 240px; position: absolute; width:
      120px" value="#{SessionBean1.ip}"/>

      <h:inputText binding="#{Page1.textField2}" id="textField2" style="height:
      25px; left: 480px; top: 240px; position: absolute; width: 144px"
      value="#{SessionBean1.puerto}"/>
```

```
        <h:outputText binding="#{Page1.outputText2}" id="outputText2" style="left:
        168px; top: 216px; position: absolute" value="Direccion IP"/>

        <h:outputText binding="#{Page1.outputText3}" id="outputText3" style="left:
        480px; top: 216px; position: absolute" value="Puerto"/>

        <h:commandLink action="Conectar" actionListener="#{SessionBean1.Conectar}"
        binding="#{Page1.linkAction1}" id="linkAction1" style="left: 360px; top:
        264px; position: absolute">
        <h:outputText binding="#{Page1.linkAction1Text}" id="linkAction1Text"
        value="Conectar"/>
       </h:commandLink>

</h:form>
```

El formulario anterior, que contiene las componentes JSF, se enmarca en la siguiente estructura de tags:

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root version="1.2" xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html" xmlns:jsp="http://java.sun.com/JSP/Page">

    <jsp:directive.page contentType="text/html;charset=UTF-8" pageEncoding="UTF-
    8"/>
    <f:view>
        <html lang="es-ES" xml:lang="es-ES">
            <head>
                <meta content="no-cache" http-equiv="Cache-Control"/>
                <meta content="no-cache" http-equiv="Pragma"/>
                <title>EasyWeb Server</title>
                <link href="resources/stylesheet.css" rel="stylesheet"
                 type="text/css"/>
            </head>
            <body style="-rave-layout: grid">

                    .... (texto)

                <h:form ... >

                    .... (código comp. JSF – pag. Anterior)

                </h:form>
            </body>
        </html>
    </f:view>
</jsp:root>
```

Esto completa el código para la página JSP, escrita íntegramente en texto plano según el formato recién descrito.

Cabe resaltar el hecho que los valores ingresados en los campos de entrada de texto 1 y 2 son asociados a los atributos `ip` (de tipo string) y `puerto` (de tipo int) del Bean `SessionBean1.java`, mediante la sintaxis:

```
value="#{SessionBean1.puerto}, value="#{SessionBean1.ip}
```

Este código se encuentra destacado en negrita en la página anterior.

Es entonces la clase `SessionBean1.java` la que, a partir de los valores antes señalados, realiza la conexión con la tarjeta Easyweb2, a través del método `Conectar()`. Pero para que esto ocurra, el link de la página de bienvenida debe, además de ser un enlace a la segunda página JSP, invocar a este método. Esto se realiza mediante la sintaxis (destacado en negrita en la página anterior):

```
actionListener="#{SessionBean1.Conectar}"
```

Gracias a esto, aunque no en forma simultánea, a través de la página de bienvenida es posible establecer conexión con varios host remotos, accediendo a una página de panel de control diferente por cada conexión establecida.

La página `panelControl.jsp` contiene las componentes JSF que permiten, una vez establecida la conexión, invocar los métodos apropiados en el Bean de sesión. Una vez ejecutados en el servidor web, estos métodos permiten el envío de comandos de control hacia el host remoto y la recepción de datos desde la tarjeta Easyweb.

Las componentes JSF presentes en la página panel de control son (fig. 4.24):

- 2 campos de entrada de texto, para el envío de caracteres al LCD de la tarjeta Easyweb2.
- 1 botón de comando para el envío del mensaje al LCD.
- 2 campos de despliegue dinámico de texto que indican el estado del status-led y del relé.
- 2 botones de comando para cambiar el estado del status-led y del relé.
- 1 botón de comando para actualizar el estado del status-led y del relé.
- 1 botón de comando para activar el buzzer.
- 1 imagen dinámica que permite el despliegue del valor porcentual de la conversión del canal 7 del ADC12 presente en el microcontrolador.
- 1 botón de comando para efectuar la adquisición del valor anterior.
- 1 área de texto, para el registro de las acciones efectuadas desde el panel de control.
- 1 botón para terminar la conexión con el host remoto.
- 2 campos de despliegue dinámico de texto que indican la IP y el puerto del host remoto con el que se ha establecido la conexión.
- 1 link a la página de bienvenida.

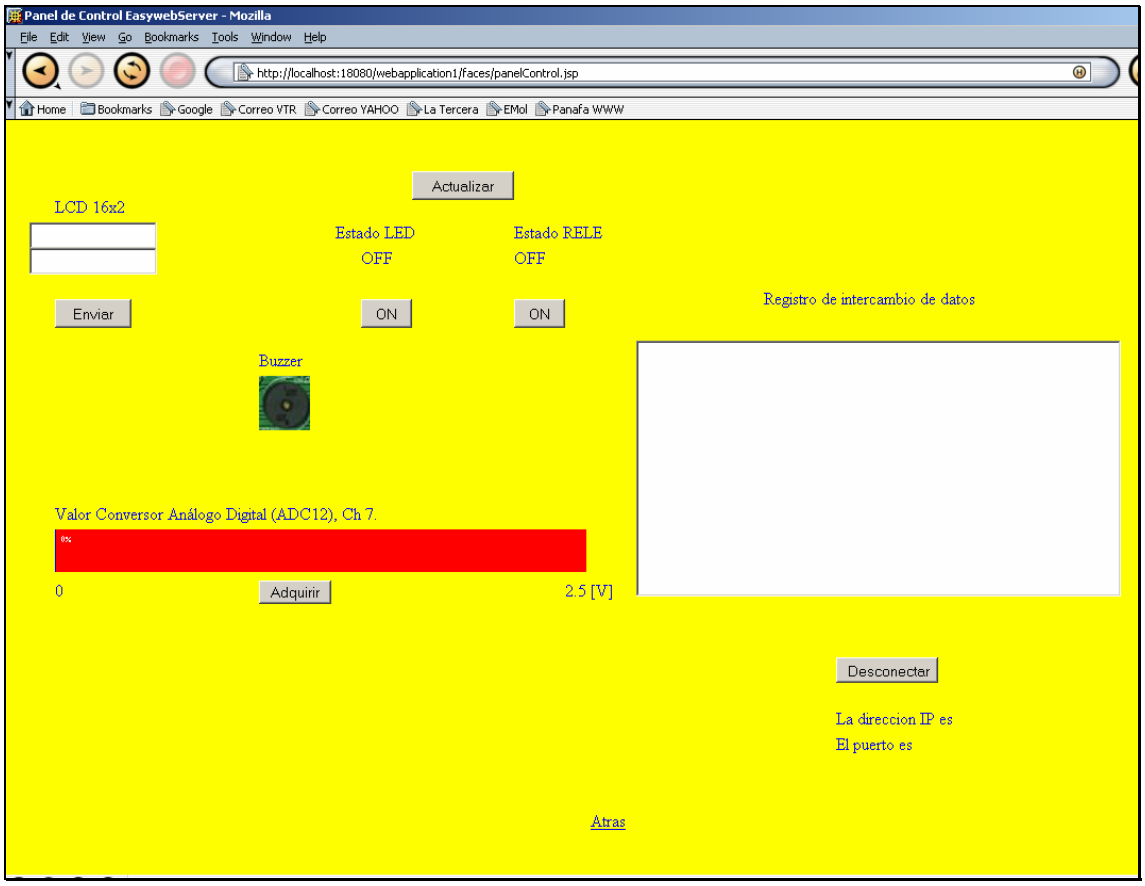


Fig. 4.24. Panel de control.

El botón de actualización del estado del relé y del led cobra sentido al considerar que el stack TCP/IP que corre en la tarjeta Easyweb2 acepta múltiples conexiones. En otras palabras, varios usuarios pueden establecer una conexión con la misma tarjeta en forma simultánea y accionar los periféricos sin que se realice una notificación automática de los cambios para todos los usuarios. Éstos deberán presionar en el botón actualizar para comprobar si otro usuario ha alterado en tiempo de sesión el estado del relé y/o del led.

El código para esta página tiene la misma estructura que la página de bienvenida. La diferencia es que esta página contiene más de un formulario, para separar por grupos las componentes JSF y sus valores. La asociación de las componentes a los atributos y métodos del Bean de sesión se realiza con la misma sintaxis antes descrita. El código de la página `panelControl.jsp` correspondiente a los distintos formularios es el siguiente:

```
<h:form binding="#{panelControl.form1}" id="form1">
       <h:outputText binding="#{panelControl.outputText3}" id="outputText3" style="left:
       984px; top: 552px; position: absolute" value="#{SessionBean1.ip}"/>

      <h:outputText binding="#{panelControl.outputText4}" id="outputText4" style="left:
      984px; top: 576px; position: absolute" value="#{SessionBean1.puerto}"/>

      <h:commandLink action="Atras" binding="#{panelControl.linkAction1}" id="linkAction1"
      style="left: 552px; top: 648px; position: absolute">
```

```
        <h:outputText binding="#{panelControl.linkAction1Text}" id="linkAction1Text"
        value="Atras"/>
       </h:commandLink>

        <h:commandButton action="#{panelControl.button1_action}"
        actionListener="#{SessionBean1.Desconectar}" binding="#{panelControl.button1}"
       id="button1" style="height: 24px; left: 840px; top: 504px; position: absolute; width:
       96px" value="Desconectar"/>

       <h:inputTextarea binding="#{panelControl.textArea1}" id="textArea1" readonly="true"
       rows="200"
       style="height: 240px; left: 648px; top: 216px; position: absolute; width: 456px"
       value="#{SessionBean1.log}"/>

       <h:inputText binding="#{panelControl.textField1}" id="textField1" maxlength="16"
       size="16"
       style="height: 25px; left: 24px; top: 96px; position: absolute; width: 120px"
       value="#{SessionBean1.LCD1}"/>

       <h:inputText binding="#{panelControl.textField2}" id="textField2" maxlength="16"
       size="16"
       style="height: 25px; left: 24px; top: 120px; position: absolute; width: 120px"
       value="#{SessionBean1.LCD2}"/>

       <h:commandButton actionListener="#{SessionBean1.buzzer}"
       binding="#{panelControl.button2}" id="button2" image="resources/buzzer.JPG"
      style="left: 240px; top: 240px; position: absolute" value="Submit"/>

        <h:commandButton actionListener="#{SessionBean1.actual}"
        binding="#{panelControl.button3}" id="button3"
       style="height: 27px; left: 384px; top: 48px; position: absolute; width: 96px"
       value="Actualizar"/>

       <h:commandButton actionListener="#{SessionBean1.lcd_send}"
       binding="#{panelControl.button7}" id="button7"
       style="height: 27px; left: 48px; top: 168px; position: absolute; width: 72px"
       value="Enviar"/>
</h:form>
<h:form>
       <h:outputText binding="#{panelControl.outputText7}" id="outputText7" style="left:
       336px; top: 120px; position: absolute" value="#{SessionBean1.accionLed1}"/>

       <h:commandButton actionListener="#{SessionBean1.switchLed}"
       binding="#{panelControl.button5}" id="button5"
       style="height: 27px; left: 336px; top: 168px; position: absolute; width: 48px"
       value="#{SessionBean1.accionLed2}"/>
</h:form>

<h:form>
       <h:outputText binding="#{panelControl.outputText8}" id="outputText8" style="left:
       480px; top: 120px; position: absolute" value="#{SessionBean1.accionRele1}"/>

       <h:commandButton actionListener="#{SessionBean1.toggleRele}"
       binding="#{panelControl.button4}" id="button4"
       style="height: 27px; left: 480px; top: 168px; position: absolute; width: 48px"
       value="#{SessionBean1.accionRele2}"/>
</h:form>

<h:form>
       <h:graphicImage binding="#{panelControl.image1}" id="image1" style="left: 48px; top:
       384px; position: absolute" value="#{SessionBean1.urlAdc}"/>

       <h:commandButton actionListener="#{SessionBean1.adcAdq}"
       binding="#{panelControl.button6}" id="button6"
      style="left: 240px; top: 432px; position: absolute" value="Adquirir"/>
</h:form>
```

Se destacan en negrita las partes del código que permiten efectuar la asociación de las componentes JSF a los métodos y atributos del `SessionBean1.java`.

Debido a la longitud del código de la clase `SessionBean1.java`, y a la similitud que presenta con las aplicaciones programadas en Java y descritas extensamente a lo largo de este trabajo (algunos métodos son esencialmente los mismos), el código completo de este Bean se entrega en el anexo. Lo mismo sucede con el código de la aplicación programada para la tarjeta de desarrollo. Salvo por la existencia de algunos casos más en la función `newdata()` (para el manejo del display y de la conversión análoga a digital), el resto de la aplicación es similar a la descrita en el punto 5.2.3.

Resta señalar que para verificar el funcionamiento y/o depurar la aplicación web programada, basta con utilizar los botones del Studio Creator señalados en la fig. 4.25. Una vez desplegada al servidor, la aplicación queda accesible a través del browser configurado por el usuario. A este respecto, es importante agregar también que algunas componentes JSF hacen uso de Javascript, lo que no necesariamente es soportado por todos los browsers. Un ejemplo de esto son las componentes Checkbox, que mostraron un correcto funcionamiento en un navegador Mozilla 1.6, no así en Internet Explorer 6.0; por esto, su uso se descartó.

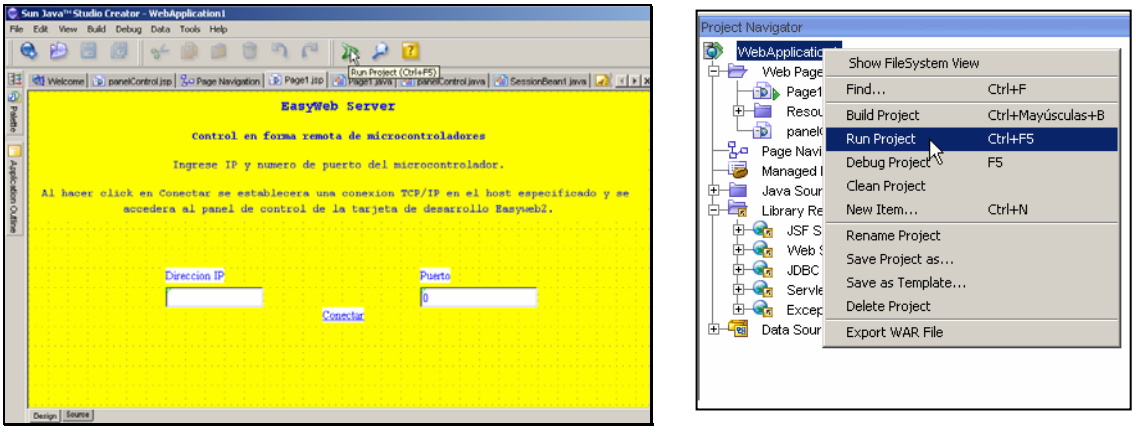


Fig. 4.25. Ejecutar aplicación web desde Studio Creator

Conclusiones

En su conjunto, este trabajo es fruto de un proceso que se inició a fines de 2003 con la llegada de las tarjetas de desarrollo Easyweb al Departamento de Electrónica de la Universidad Santa María. Un primer trabajo (consultar [16]), se enfocó a la descripción de las posibilidades de uso de la tarjeta en el marco de un curso de introducción al diseño con microcontroladores y se sugirió la exploración de las posibilidades de conectividad en red de la tarjeta. El curso efectivamente se realizó y surgieron interesantes proyectos, algunos de ellos referenciados en este trabajo [26].

Un objetivo tácito de este trabajo ha sido aunar todo el conjunto de experiencias acumuladas hasta acá en relación a las posibilidades de conexión a redes de computadores de la tarjeta Easyweb. Así como el trabajo aludido sirvió como material de apoyo para el desarrollo del curso antes mencionado, siempre fue la intención del autor que el material aquí presentado sea de utilidad y oriente al estudiante programador de microcontroladores MSP430 que desee incorporar en su proyecto la conectividad a redes IP. Es por esto que el desarrollo de las aplicaciones sigue un hilo conductor en el que se abordan desde los programas más simples posibles hasta aplicaciones más complejas, pero que no por ello son lejanas para un programador en C de nivel medio. Se trata, en esencia, de la síntesis documentada de un método estructurado para la programación de aplicaciones de red sobre el stack uIP [14], para el microcontrolador MSP430F149 y su interfaz de red.

Quiso incluirse, en forma paralela, la programación de aplicaciones de usuario en un lenguaje orientado a objeto, como es Java. Esto permitió entregar un enfoque distinto al trabajo, que podría haberse limitado a la sola programación de microcontroladores. Además de constituir un desafío, puesto que no se contaba con experiencia previa al respecto, el hecho de contar con aplicaciones que verificaran de una manera amigable las posibilidades de conexión a la tarjeta de desarrollo, ejecutables desde un PC de usuario, se convirtió en una necesidad puesto que una aplicación que cuenta con una interfaz gráfica es mucho más cercana para el usuario común. En este mismo contexto, y gracias a la exploración de las posibilidades de la tecnología de programación Java, surge en la etapa final del trabajo la idea de proponer un nuevo esquema para acceder al control y a la adquisición de datos desde la tarjeta a través de un browser.

Con el objetivo, expuesto extensamente en el capítulo 4, de liberar recursos del microcontrolador y así dejar en memoria sólo las funciones de red y la aplicación de control respectiva, se programan aplicaciones web para su ejecución en un servidor de aplicaciones. Para esto se utilizan servlets, plantillas JSP y la reciente implementación JSF, para la creación de componentes web. El ambiente de desarrollo utilizado, Sun Java Studio Creator 2004Q2 [33], es uno de los más recientes programas para el desarrollo de aplicaciones web, liberado por la Sun a mediados del presente año. El resultado es una aplicación que constituye un real aporte en el área de la conectividad web de microcontroladores, especialmente por la modularización del proceso de programación: las aplicaciones pueden ser desarrolladas por un equipo integrado por programadores de microcontroladores en lenguaje de bajo nivel, programadores Java y desarrolladores Web.

Trabajos Futuros

El campo para el desarrollo de futuros trabajos en esta área queda totalmente abierto. Desde luego, todo lo que tiene que ver con la seguridad en el establecimiento de las conexiones y la transferencia de los datos no se trató en este trabajo. Si incluyera un sistema de autentificación de usuario, más el uso de encriptación a través de un protocolo seguro (https, por ejemplo), la aplicación final podría constituir perfectamente un producto para el control en forma remota de artefactos del hogar. O bien ser una interfaz para el monitoreo en forma remota de un proceso simple. Al pensar en el control de procesos industriales, se abre todo un campo de investigación, en el que siguen emergiendo las bases teóricas para enfrentar problemas como la estabilidad en la operación de sistemas de control de tiempo real interconectados y los efectos de las redes en el desempeño de los sistemas de control [35]. Se trata de una convergencia tecnológica entre las telecomunicaciones, el control automático y la computación.

Otra posibilidad que queda sin haber sido abordada es la interconexión de varios microcontroladores entre sí, para la creación de una red local de sistemas embebidos.

No obstante, la arquitectura de red propuesta, que incluye un servidor web como servidor de aplicaciones, deja el camino abierto para que estas ideas se integren a futuros proyectos de una manera simple y mantiene al sistema embebido libre de cargas adicionales.

Finalmente, cabe señalar que todos los códigos de las aplicaciones desarrolladas a lo largo de este trabajo se adjuntan en un CD a este documento impreso, para su reutilización por parte de estudiantes o de quien desee desarrollar aplicaciones para la conectividad IP de microcontroladores.

Referencias

[1] "Computer Organization and Design", 3rd Edition. Patterson, D., Hennessy, J.

[2] IPSIL http://www.ipsil.com

[3] "Introduction to TCP/IP, part 1". Embedded Systems Programming, Internet Appliance Design. Herbert, T. Disponible en: http://www.embedded.com/internet/9912/9912ia1.htm

[4] "IP & Ethernet Interfaces", Beyond Logic. Disponible en: http://www.beyondlogic.org/etherip/ip.htm

[5] KwikNet Internet Acces Company. http://www.kwiknet.com

[6] SMX RTOS. http://www.smxinfo.com/

[7] Fusion™ Embedded TCP/IP Stack. http://www.unicoi.com/fusion_net/fusion_tcpip.htm

[8] Interpeak. http://www.interpeak.com/

[9] NetBurner. http://www.netburner.com/products/software_and_protocols/tcp_ip_stack.htm

[10] Blunk Microsystems. http://www.blunkmicro.com/tcp.htm

[11] CMX Systems – Micronet. http://www.cmx.com/micronet.htm

[12] uCLinux. http://www.beyondlogic.org/uClinux/uClinux.htm

[13] OpenTCP Project. http://www.opentcp.org/index.php

[14] uIP TCP/IP Stack for Embedded Microcontrollers. Dunkels, A. http://www.sics.se/~adam/uip/

[15] lwIP A Lightweight TCP/IP stack. http://savannah.nongnu.org/projects/lwip/

[16] "Técnicas de Diseño con Microcontrolador MSP430", Zúñiga, J. Memoria para optar al título de Ingeniero Civil Electrónico. Profesor Guía: Silva, L. 2003.

[17] "MSP430x1xx Family User's Guide", Texas Instruments, 2003. Disponible en: http://www.ti.com/msp430

[18] MSP430x13x, MSP430x14x, MSP430x14x1. Mixed Signal Microcontroller. (SLAS272e). Disponible en http://www.ti.com/msp430

[19] cs8900A Product Datasheet. Disponible en: http://www.cirrus.com/en/products/pro/detail/P46.html

[20] CS8900A FAQs. Disponible en: http://www.cirrus.com/en/products/pro/detail/P46.html

[21] MSP430 Internet Connectivity. Disponible en: http://ww.softbaugh.com/downloads/slaa137.pdf

[22] MSP430 Port by Paul Curtis. http://www.rowley.co.uk/msp430/uip.htm

[23] IAR Embedded Workbench EW430. http://www.iar.com/Products/?name=EW430

[24] Tutorial de emulación con IAR Embedded Workbench EW430. Naveas, P. Disponible en: http://www.elo.utfsm.cl/~lsb/elo325/elo325.html

[25] Tutorial: Programación en red. Gómez, F. http://www.programacion.com/java/tutorial/joa_red/1/

[26] "Proyecto Datalogger". Valencia, D., Kusch, M., Blancaire, F., González, E. Seminario de Diseño con Microcontroladores. Primer semestre 2004, UTFSM. Información en: http://www.elo.utfsm.cl/~lsb/elo325/elo325.html

[27] Java 2 Platform Standard Edition v. 1.4.2 Download. http://java.sun.com/j2se/1.4.2/download.jsp

[28] GUI Building in NetBeans IDE 3.6. http://www.netbeans.org/kb/articles/form_getstart36.html

[29] Design & Development of simple Java[tm] servlet applications. https://learningcenter-sai.sun.com/gui/ito/generic_index.jsp

[30] J2EE JavaServer Pages Technology. http://java.sun.com/products/jsp/

[31] JavaBeans Components. The J2EE 1.4 Tutorial, Ch. 12. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

[32] J2EE JavaServer Faces. http://java.sun.com/j2ee/javaserverfaces/index.jsp

[33] Sun Java Studio Creator 2004Q2. http://wwws.sun.com/software/products/jscreator/index.html

[34] JSFCentral Product Directory. http://www.jsfcentral.com/products/

[35] Guest Editorial "Special Issue on Networked Control Systems". Antsaklis, P. Baillieul, J. IEEE Transactions on Automatic Control, vol. 49, nº 9, sept. 2004.

Anexo.
Códigos de aplicaciones.

Aplicación cliente SMTP (funciones newdata, acked y senddata)

```
* SMTP example implementation
* Copyright (c) 2002, Adam Dunkels.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * This file is part of the uIP TCP/IP stack.
 *
 * $Id: smtp.c,v 1.1.2.7 2003/10/07 13:47:50 adam Exp $
 */
#define ISO_nl 0x0a
#define ISO_cr 0x0d
#define ISO_period 0x2e
#define ISO_2  0x32
#define ISO_3  0x33
#define ISO_4  0x34
#define ISO_5  0x35

/* 220 */
char smtp_220[4] = {0x32, 0x32, 0x30, };
/* HELO  */
char smtp_helo[6] = {0x48, 0x45, 0x4c, 0x4f, 0x20, };
/* MAIL FROM:  */
char smtp_mail_from[12]={0x4d,0x41, 0x49, 0x4c, 0x20, 0x46, 0x52, 0x4f, 0x4d, 0x3a, 0x20, };
/* RCPT TO:  */
char smtp_rcpt_to[10] = {0x52, 0x43, 0x50, 0x54, 0x20, 0x54, 0x4f, 0x3a, 0x20, };
/* DATA */
char smtp_data[5] = {0x44, 0x41, 0x54, 0x41, };
/* To:  */
char smtp_to[5] = {0x54, 0x6f, 0x3a, 0x20, };
/* From:  */
char smtp_from[7] = {0x46, 0x72, 0x6f, 0x6d, 0x3a, 0x20, };
/* Subject:  */
char smtp_subject[10] = {0x53, 0x75, 0x62, 0x6a, 0x65, 0x63, 0x74, 0x3a, 0x20, };
/* QUIT */
char smtp_quit[5] = {0x51, 0x55, 0x49, 0x54, };

static void
newdata(struct smtp_state *s)
{
  //Si llega 5** es comando ilegal para el servidor smtp
  if(*(char *)uip_appdata == ISO_5) {
    smtp_done(1);
    uip_abort();
    return;
  }
      uip_appdata);
  switch(s->state) {
  case STATE_SEND_NONE:
    if(strncmp((char *)uip_appdata, smtp_220, 3) == 0) {
      s->state = STATE_SEND_HELO;
      s->sendptr = 0;
    }
    break;

  case STATE_SEND_HELO:

    if(*(char *)uip_appdata == ISO_2) {
      s->state = STATE_SEND_MAIL_FROM;
      s->sendptr = 0;
    }
    break;
```

```
  case STATE_SEND_MAIL_FROM:
    if(*(char *)uip_appdata == ISO_2) {
      s->state = STATE_SEND_RCPT_TO;
      s->textlen = s->sendptr = 0;
    }
    break;

  case STATE_SEND_RCPT_TO:
    if(*(char *)uip_appdata == ISO_2) {
      s->state = STATE_SEND_DATA;
      s->textlen = s->sendptr = 0;
    }
    break;

  case STATE_SEND_DATA:
    if(*(char *)uip_appdata == ISO_3) {
      s->state = STATE_SEND_DATA_HEADERS;
      s->textlen = s->sendptr = 0;
    }
    break;
  case STATE_SEND_DATA_HEADERS:
    if(*(char *)uip_appdata == ISO_3) {
      /*      printf("3\n");*/
      s->state = STATE_SEND_DATA_MESSAGE;
      s->textlen = s->sendptr = 0;
    }
    break;
  }
}

static void
acked(struct smtp_state *s)
{
  s->sendptr += s->sentlen;
  s->sentlen = 0;

  if(s->sendptr == s->textlen) {
    switch(s->state) {
    case STATE_SEND_DATA_HEADERS:
      s->state = STATE_SEND_DATA_MESSAGE;
      s->sendptr = s->textlen = 0;
      break;
    case STATE_SEND_DATA_MESSAGE:
      s->state = STATE_SEND_DATA_END;
      s->sendptr = s->textlen = 0;
      break;
    case STATE_SEND_DATA_END:
      s->state = STATE_SEND_QUIT;
      s->sendptr = s->textlen = 0;
      break;
    case STATE_SEND_QUIT:
      s->state = STATE_SEND_DONE;
      smtp_done(SMTP_ERR_OK);
      uip_close();
      break;
    }
  }
}


//envio de datos
static void
senddata(struct smtp_state *s)
{
  char *textptr;

  if(s->textlen != 0 &&
     s->textlen == s->sendptr) {
    return;
  }
```

```
textptr = (char *)uip_appdata;
switch(s->state) {
case STATE_SEND_HELO:
  /* Create HELO message. */
  strcpy(textptr, smtp_helo);
  textptr += sizeof(smtp_helo) - 1;
  strcpy(textptr, localhostname);
  textptr += strlen(localhostname);
  *textptr = ISO_cr;
  ++textptr;
  *textptr = ISO_nl;
  ++textptr;
  /*    printf("s->sendptr %d\n", s->sendptr);*/
  if(s->sendptr == 0) {
    s->textlen = textptr - (char *)uip_appdata;
    /*      printf("s->textlen %d\n", s->textlen);*/
  }
  textptr = (char *)uip_appdata;
  break;
case STATE_SEND_MAIL_FROM:
  /* Create MAIL FROM message. */
  strcpy(textptr, smtp_mail_from);
  textptr += sizeof(smtp_mail_from) - 1;
  strcpy(textptr, s->from);
  textptr += strlen(s->from);
  *textptr = ISO_cr;
  ++textptr;
  *textptr = ISO_nl;
  ++textptr;
  if(s->sendptr == 0) {
    s->textlen = textptr - (char *)uip_appdata;
  }
  textptr = (char *)uip_appdata;
  break;
case STATE_SEND_RCPT_TO:
  /* Create RCPT_TO message. */
  strcpy(textptr, smtp_rcpt_to);
  textptr += sizeof(smtp_rcpt_to) - 1;
  strcpy(textptr, s->to);
  textptr += strlen(s->to);
  *textptr = ISO_cr;
  ++textptr;
  *textptr = ISO_nl;
  ++textptr;
  if(s->sendptr == 0) {
    s->textlen = textptr - (char *)uip_appdata;
  }
  textptr = (char *)uip_appdata;
  break;
case STATE_SEND_DATA:
  strcpy(textptr, smtp_data);
  textptr += sizeof(smtp_data) - 1;
  *textptr = ISO_cr;
  ++textptr;
  *textptr = ISO_nl;
  ++textptr;
  if(s->sendptr == 0) {
    s->textlen = textptr - (char *)uip_appdata;
  }
  textptr = (char *)uip_appdata;
  break;
case STATE_SEND_DATA_HEADERS:
  /* Create mail headers-> */
  strcpy(textptr, smtp_to);
  textptr += sizeof(smtp_to) - 1;
  strcpy(textptr, s->to);
  textptr += strlen(s->to);
  *textptr = ISO_cr;
  ++textptr;
  *textptr = ISO_nl;
  ++textptr;
```

```
    strcpy(textptr, smtp_from);
    textptr += sizeof(smtp_from) - 1;
    strcpy(textptr, s->from);
    textptr += strlen(s->from);
    *textptr = ISO_cr;
    ++textptr;
    *textptr = ISO_nl;
    ++textptr;

    strcpy(textptr, smtp_subject);
    textptr += sizeof(smtp_subject) - 1;
    strcpy(textptr, s->subject);
    textptr += strlen(s->subject);
    *textptr = ISO_cr;
    ++textptr;
    *textptr = ISO_nl;
    ++textptr;

    if(s->sendptr == 0) {
      s->textlen = textptr - (char *)uip_appdata;
    }
    textptr = (char *)uip_appdata;
    break;
  case STATE_SEND_DATA_MESSAGE:
    textptr = s->msg;
    if(s->sendptr == 0) {
      s->textlen = s->msglen;
    }
    break;
  case STATE_SEND_DATA_END:
    *textptr = ISO_cr;
    ++textptr;
    *textptr = ISO_nl;
    ++textptr;
    *textptr = ISO_period;
    ++textptr;
    *textptr = ISO_cr;
    ++textptr;
    *textptr = ISO_nl;
    ++textptr;
    if(s->sendptr == 0) {
      s->textlen = 5;
    }
    textptr = (char *)uip_appdata;
    break;
  case STATE_SEND_QUIT:
    strcpy(textptr, smtp_quit);
    textptr += sizeof(smtp_quit) - 1;
    *textptr = ISO_cr;
    ++textptr;
    *textptr = ISO_nl;
    ++textptr;
    if(s->sendptr == 0) {
      s->textlen = textptr - (char *)uip_appdata;
    }
    textptr = (char *)uip_appdata;
    break;
  default:
    return;
  }
  textptr += s->sendptr;

  if(s->textlen - s->sendptr > uip_mss()) {
    s->sentlen = uip_mss();
  } else {
    s->sentlen = s->textlen - s->sendptr;
  }

  uip_send((u8_t *)textptr, s->sentlen);
}
```

Aplicación web final.

```
/*
 * SessionBean1.java
 *
 * Created on 19 de octubre de 2004, 8:45
 * Copyright KUKO
 */
package webapplication1;

import javax.faces.*;
import com.sun.jsfcl.app.*;
import javax.faces.event.ActionEvent;
import javax.faces.event.ValueChangeEvent;
import javax.faces.context.FacesContext;
import java.io.*;
import java.net.*;



public class SessionBean1 extends AbstractSessionBean {

    public String ip = null;
    public Integer puerto = null;

    public String LCD1 = null;
    public String LCD2 = null;

    private StringBuffer logMsgs = new StringBuffer();

    private boolean ifLedOn;
    private boolean ifReleOn;

    String adcValue = "0";

    private Socket mspSocket;
    private DataOutputStream outToServer;
    private BufferedReader inFromServer;
    private String uipMsg = "default";



    // <editor-fold defaultstate="collapsed" desc="Creator-managed Component Definition">

    private int __placeholder;

    // </editor-fold>
    public SessionBean1() {
   // <editor-fold defaultstate="collapsed" desc="Creator-managed Component Initialization">
        try {
        } catch (Exception e) {
            log("SessionBean1 Initialization Failure", e);
            throw e instanceof FacesException ? (FacesException) e : new FacesException(e);
        }
        // </editor-fold>
        // Additional user provided initialization code
    }

    protected webapplication1.ApplicationBean1 getApplicationBean1() {
        return (webapplication1.ApplicationBean1)getBean("ApplicationBean1");
    }

    /**
     * Bean cleanup.
     */
    protected void afterRenderResponse() {
    }
```

```
public void getLedStatus() {
    /*Perfectamente podria ir aqui una rutina q invocara el estado real del Rele*/
    //return true;  //Lo suponemos siempre cerrado al inicio
    try {
        outToServer.writeChar('1');
        uipMsg = inFromServer.readLine();
    }
    catch (IOException e) {
        System.out.println( e );
    }
    if (uipMsg.equals("1")){
        this.setIfLedOn(true);
        //addLogMsg("Cambio ifLedOn a verdadero (recibo 1)");
    }
    else {
        this.setIfLedOn(false);
        //addLogMsg("Cambio ifLedOn a falso (recibo 0)");
    }
}

public void getReleStatus() {
    /*Perfectamente podria ir aqui una rutina q invocara el estado real del Rele*/
    //return true;  //Lo suponemos siempre cerrado al inicio
    try {
        outToServer.writeChar('2');
        uipMsg = inFromServer.readLine();
    }
    catch (IOException e) {
        System.out.println( e );
    }
    if (uipMsg.equals("1")){
        this.setIfReleOn(true);
        //addLogMsg("Cambio ifReleOn a verdadero (recibo 1)");
    }
    else {
        this.setIfReleOn(false);
        //addLogMsg("Cambio ifReleOn a falso (recibo 0)");
    }
}
public void Conectar(ActionEvent ae) {
    connect(this.getIp(), this.getPuerto().intValue());
}

/**
 * Getter for property ip.
 * @return Value of property ip.
 */
public String getIp() {
    return this.ip;
}

/**
 * Setter for property ip.
 * @param ip New value of property ip.
 */
public void setIp(String ip) {
    this.ip = ip;
}

/**
 * Getter for property puerto.
 * @return Value of property puerto.
 */
public Integer getPuerto() {
    return this.puerto;
}

/**
 * Setter for property puerto.
 * @param puerto New value of property puerto.
 */
```

```
    public void setPuerto(Integer puerto) {
        this.puerto = puerto;
    }

    public void connect(String ip, int port) {

        try {
            mspSocket = new Socket(ip, port);
            outToServer =  new DataOutputStream(mspSocket.getOutputStream());
            inFromServer = new BufferedReader(new
InputStreamReader(mspSocket.getInputStream()));
            uipMsg = inFromServer.readLine();
        }
        catch (IOException e) {
            System.out.println( e );
        }
        //System.out.println("Conectando a "+ip+" en el puerto "+String.valueOf(port));
        try{
            logMsgs.delete(0, logMsgs.length());
        }catch(Exception e){
            System.out.println("Error");
        }
        addLogMsg("Conectado a "+ip+" en el puerto "+String.valueOf(port)+'\n');
        addLogMsg(uipMsg);
        getLedStatus();
        getReleStatus();
    }

    public String getLog() {
        return this.logMsgs.toString();
    }

    public void addLogMsg(String msg) {
        this.logMsgs = new
StringBuffer().append(this.logMsgs.toString()).append('\n').append(msg);
    }

    public void switchLed(ActionEvent ae) {
        this.ifLedOn = !this.ifLedOn;
        this.Led1ON();
    }

    public void setIfLedOn(boolean b) {
        ifLedOn = b;
    }

    public boolean getIfLedOn() {
        return ifLedOn;
    }

    public String getAccionLed1() {
        if(!this.getIfLedOn())
            return "OFF";
        return "ON";
    }

    public String getAccionLed2() {
        if(!this.getIfLedOn())
            return "ON";
        return "OFF";
    }

    public String getUrlAdc() {
        if (adcValue.length()==3){
            String adcValueOk = adcValue.substring(1);
            return ("http://www.telematica.utfsm.cl/temp/ima.php?pe=" + adcValueOk);
        }
        else return("http://www.telematica.utfsm.cl/temp/ima.php?pe=" + adcValue);
    }
```

```
  public void Led1ON() {
      try {
          outToServer.writeChar('B');
      }
      catch (IOException e) {
          System.out.println( e );
      }
      addLogMsg("Led en "+String.valueOf(getIfLedOn()));
  }

  public void Desconectar(ActionEvent ae) {
      disconnect();
      this.setIp(null);
      this.setPuerto(null);
  }

  public void disconnect() {
      try {
          outToServer.writeChar('F');
          uipMsg = inFromServer.readLine();
          mspSocket.close();
      }
      catch (IOException e) {
          System.out.println( e );
      }
      addLogMsg(uipMsg + '\n');
      addLogMsg("Haz click en atras para conectar nuevamente");
  }

  public void ActivarRele(ActionEvent ae) {
      Rele1ON();
  }

/*
 * Getter & Setter Methods para las variables de estado
 */
  public void setIfReleOn(boolean b) {
      ifReleOn = b;
  }

  public boolean getIfReleOn() {
      return ifReleOn;
  }

  public void toggleRele(ActionEvent ae) {
      this.ifReleOn = !this.ifReleOn;
      this.Rele1ON();
  }

  public String getAccionRele1() {
      if(!this.getIfReleOn())
          return "OFF";
      return "ON";
  }

  public String getAccionRele2() {
      if(!this.getIfReleOn())
          return "ON";
      return "OFF";
  }

  public void Rele1ON() {
      try {
          outToServer.writeChar('C');
      }
      catch (IOException e) {
          System.out.println( e );
      }
      addLogMsg("Rele en "+String.valueOf(getIfReleOn()));
  }
```

```
    public void lcd_send(ActionEvent ae) {
        try {
            outToServer.writeChar('A');
            outToServer.writeBytes(LCD1 + LCD2);
        }
        catch ( IOException e ) {
            System.out.println( e );}
        addLogMsg("Mensaje enviado a LCD");
    }

    /**
     * Getter for property LCD1.
     * @return Value of property LCD1.
     */
    public String getLCD1() {
        return this.LCD1;
    }

    /**
     * Setter for property LCD1.
     * @param LCD1 New value of property LCD1.
     */
    public void setLCD1(String LCD1) {
        this.LCD1 = LCD1;
    }

    /**
     * Getter for property LCD2.
     * @return Value of property LCD2.
     */
    public String getLCD2() {
        return this.LCD2;
    }

    /**
     * Setter for property LCD2.
     * @param LCD2 New value of property LCD2.
     */
    public void setLCD2(String LCD2) {
        this.LCD2 = LCD2;
    }

    public void buzzer(ActionEvent ae) {

        try{ outToServer.writeChar('E'); }
        catch ( IOException e ) {
            System.out.println( e );}
        addLogMsg("Buzzer activado");
    }

    public void adcAdq(ActionEvent ae) {
        try {
            outToServer.writeChar('G');
            adcValue = "0";
            adcValue = inFromServer.readLine();
        }
        catch ( IOException e ) {
            System.out.println( e );}
    }

    public void actual(ActionEvent ae) {
        getLedStatus();
        getReleStatus();
    }
}
```

```
/*
 * appweb2.c
 *
 * Copyright KUKO
 */

#include <string.h>
#include <msp430x14x.h>
#include "uip.h"
#include "appweb2.h"
#include "ports_LCD.c" //configuracion pines I/O y LCD.

#define LED_STATE   '1'
#define RELE_STATE  '2'
#define LCD         'A'
#define STATUS_LED  'B'
#define RELE1       'C'
#define RELE2       'D'
#define BUZZER      'E'
#define FIN         'F'
#define ADC         'G'


#define NO_CONECTADO 0
#define CONECTADO    1
#define SALIR        2


unsigned char welcome[31] = "EASYWEB: Conexion Establecida.\n";
unsigned char exit[7] = "Adios!\n";
unsigned char led_rele_state_on[2] = "1\n";
unsigned char led_rele_state_off[2] = "0\n";


//inicializacion de la aplicacion
void appweb2_init(void) {
   struct appweb2_struct *s = (struct appweb2_struct *)uip_conn->appstate;

   InitPorts();
   Init_Display();
   s->state = NO_CONECTADO;
   uip_listen(HTONS(1510));         //escucha por conexiones
}

//conexion establecida
static void connected (struct appweb2_struct *s) {
  s->state = CONECTADO;
  s->textptr = welcome;
  s->textlen = sizeof(welcome);
}


//administrador de eventos
void call_appweb2(void) {

  struct appweb2_struct *s;
  s = (struct appweb2_struct *)uip_conn->appstate;

  if(uip_aborted()) {
    aborted();
  }
  if(uip_timedout()) {
    timedout();
  }
  if(uip_closed()) {
    closed();
  }
  if(uip_connected()) {
    connected(s);
    senddata();
  }
```

```
  if(uip_acked()) {
    acked(s);
  }
  if(uip_newdata()) {
    newdata(s);
    senddata(s);
  }
}

static void aborted(struct appweb2_struct *s) {
  s->state = NO_CONECTADO;
}
static void timedout(struct appweb2_struct *s) {
  s->state = NO_CONECTADO;
}
static void closed(struct appweb2_struct *s) {
  s->state = NO_CONECTADO;
}

//ACK recibido
static void acked(struct appweb2_struct *s) {
  s->textlen -= uip_conn->len;
  s->textptr += uip_conn->len;
  if (s->textlen == 0) {
   switch (s->state) {
    case CONECTADO:
     break;
    case SALIR:
     uip_close();
     break;
    default:
     break;
   }
  }
}

// recepción de comando para activación de perifericos
static void newdata (struct appweb2_struct *s) {

  int ct = 0;
  int i = 0;
  int adcValue = 0;
  char AD7[5];

  switch (*(char *)uip_appdata) {

   case LED_STATE:
    if ((BIT1&P2OUT)==0) {
     s->textptr = led_rele_state_on;
     s->textlen = sizeof(led_rele_state_on);
    }
    else {
     s->textptr = led_rele_state_off;
     s->textlen = sizeof(led_rele_state_off);
    }
    break;

    case RELE_STATE:
    if ((BIT5&P1OUT)==0) {
     s->textptr = led_rele_state_off;
     s->textlen = sizeof(led_rele_state_off);
    }
    else {
     s->textptr = led_rele_state_on;
     s->textlen = sizeof(led_rele_state_on);
    }
    break;

   case LCD:
    SEND_CMD(CLR_DISP);
    SEND_CMD(LINEA1);
```

```
    for (i=1; i < 33; i++) {
     SEND_CHAR((*(char *)(uip_appdata + i)));
     if ( i == 16) SEND_CMD(LINEA2);
    }
    break;

   case STATUS_LED:
    P2OUT ^= BIT1;
    break;

   case RELE1:
    P1OUT ^= BIT5;
    break;

   case RELE2:
    P1OUT ^= BIT6;
    break;

   case BUZZER:
        while (ct < 100) {
      SEND_CMD(CLR_DISP);
         BUZ1_OFF;
      BUZ2_ON;
      Delay(100);        //buzzer 5 000 Hz
      BUZ2_OFF;
      BUZ1_ON;
      Delay(100);
         ct ++;
       }
    break;

   case ADC:
    adcValue = GetAD7Val();
    sprintf(AD7, "%d\n", (char)adcValue);
    s->textptr = AD7;
    s->textlen = sizeof(AD7)-1;
    break;

   case FIN:
    s->textptr = exit;
    s->textlen = sizeof(exit);
    s->state = SALIR;
    break;

   default:
    break;
 }
}

//envio de datos
static void senddata(struct appweb2_struct *s) {
  if(s->textlen > 0) {
    uip_send(s->textptr, s->textlen);
  }
}

int GetAD7Val(void)
{
  ADC12CTL0 = ADC12ON | SHT0_15 | REF2_5V | REFON;   // ADC on, int. ref. on (2,5 V),
                                                     // single channel single conversion
  ADC12CTL1 = ADC12SSEL_2 | ADC12DIV_7 | CSTARTADD_0 | SHP;// MCLK / 8 = 1 MHz

  ADC12MCTL0 = SREF_1 | INCH_7;                 // int. ref., channel 7

  ADC12CTL0 |= ENC;                             // enable conversion
  ADC12CTL0 |= ADC12SC;                         // sample & convert
  while (ADC12CTL0 & ADC12SC);                  // wait until conversion is complete
  ADC12CTL0 &= ~ENC;                            // disable conversion
  return ADC12MEM0/ 41;
}
```