Capítulo 4

Implementación

En este capítulo se hace una descripción de la implementación del estándar IEEE 1451 realizada. Para llevarla a cabo, se decidió seguir el IEEE 1451.0 para el formato de los TEDS y los mensajes intercambiados entre TIM y NCAP, así como para la definición comandos y funcionalidades. Como capa física, se utilizó Bluetooth, que es una de las pocas que no requiere revisión para ser adaptada a este nuevo IEEE 1451.0 debido a que ha sido sido aprobada recientemente.

4.1 Plataforma hardware

Para la implementación del NCAP se utilizó un PC portátil con sistema operativo Linux. Esto permite que la aplicación sea fácilmente portable a otras plataformas con soporte para este sistema operativo, como cualquier sistema tipo *embedded*.

Para el TIM se utilizó un microcontrolador ATmega8 junto con un módulo Bluegiga para las comunicaciones Bluetooth. En los siguientes apartados se hace una pequeña descripción de estos dos dispositivos.

Microcontrolador ATmega8

Este microcontrolador [1] pertenece a la familia AVR, una arquitectura *harvard* modificada RISC de 8 bits desarrollada por Atmel en 1996. Fue una de las primeras familias de microcontroladores en usar memoria flash integrada para el alamacenamiento del programa, en lugar de memorias de tipo EPROM o EEPROM.

La arquitectura harvard modificada implica que programa y datos son almacenadas en distintas memorias físicas, pero se tiene la capacidad de leer datos almacenados en la memoria de programa a través de instrucciones especiales. Por otro lado, la mayoría de instrucciones requieren sólo uno o dos ciclos como tiempo de ejecución, lo que convierte a los AVRs en microcontroladores relativamente rápidos entre los de 8 bits. Además, fueron diseñados teniendo en cuenta la ejecución eficiente de código en C, teniendo varios punteros integrados para esta tarea, ya que este lenguaje hace un uso profuso de ellos para el manejo de variables en memoria.

A continuación se citan algunas de las características más destacables del ATmega8:

- 8 kB de memoria flash para programa
- 512 B de memoria EEPROM

- 1024 B de memoria SRAM
- 32 registros de propósito general
- multiplicador hardware
- frecuencia de reloj de 0 a 16 MHz
- 32 pines de entrada/salida
- 2 contradores de 8 bits y otro de 16 bits
- convertidores Analógico/Digital de varios canales
- *In-System Programming* a través de SPI
- Full Duplex SPI
- interfaz serie de 2 hilos (compatible con I2C)
- Full Duplex USART

Bluegiga WT12

Se trata de un módulo Bluetooth 2.1 + EDR de clase 2 [2], conteniendo todos los elementos necesarios, desde el sistema radio Bluetooth hasta la antena, además de una pila de protocolos iWRAP. Algunas de sus características son las siguientes:

- sistema Bluetooth v2.0 + EDR completamente aprobado
- antena integrada
- compatibilidad con USB 2.0
- soporte para UART
- coexistencia con 802.11
- 8 Mbits de memoria flash

En la figura 4.2 se puede ver un diagrama de bloques de este módulo. El que resulta de más interés es el Bluecore04, una solución Blueooth integrada en un solo chip que implemente un transceptor de radio Bluetooth y un microcontrolador. Este microcontrolador actúa como controlador de interrupciones y temporizador, ejecutando además la pila software de Bluetooth y controlado el interfaz radio.

La arquitectura software del WT12 permite que el procesado de los protocolos Bluetooth y el programa de aplicación se reparta de diferentes formas entre el microcontrolador interno y el procesador externo (si existe). Las capas superiores de la pila Bluetooth (por encima del HCI) pueden ser ejecutadas en este microcontrolador o externamente.

En este proyecto se trabajó con la pila iWRAP que es la que se muestra en la figura 4.1. Con esta pila no se re quiere ningún procesador externo para ejecutar parte de la pila Bluetooth. Todas las capas software se ejecutan en el microcontrolador interno del dispositivo.

El procesador externo, en este caso el ATmega8, se comunicac on el software iWRAP a través de uno o más de los diferentes interfaces físicos. La forma más común es la comunicación vía UART

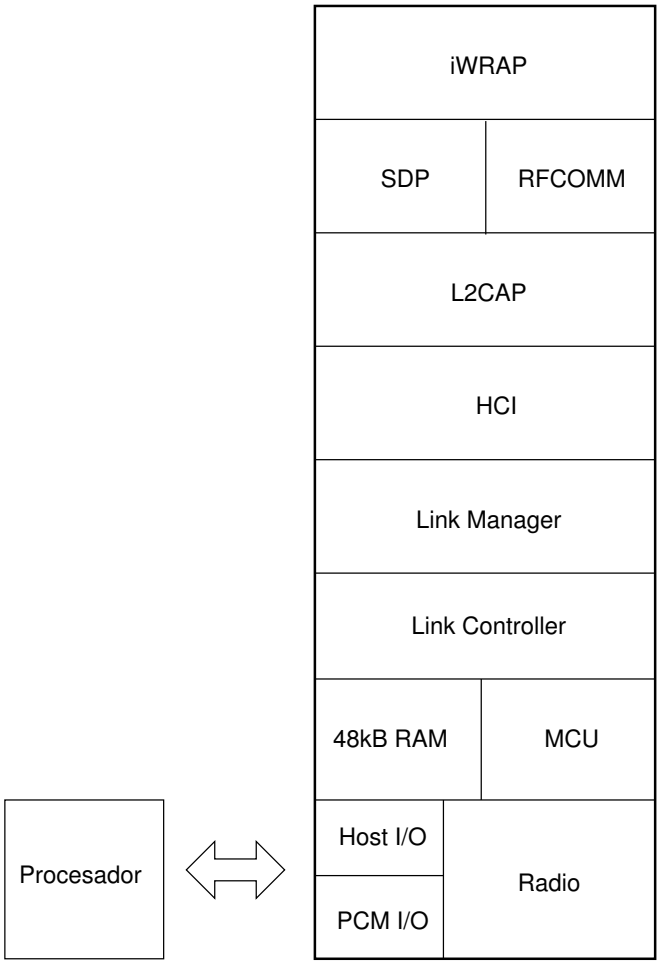

Figura 4.1: Pila de protocolos iWRAP

a través de los diferentes comandos ASCII soportados por el software iWRAP, tal como se hace en la placa utilizada en este proyecto. A través de estos comandos se puede acceder a la funcionalidad Bluetooth sin prestar atención a la complejidad de toda la pila.

4.2 Plataforma software

Para llevar a cabo la programación de todos los dispositivos se utilizaron herramientas de software libre, que son las que se comentan a continuación.

Binutils-AVR

Este paquete [3] proporciona todas as utilidades necesarias para construir y manipular código objeto. Una vez instalado, se cuenta con utilidades tales como un ensamblador AVR (avr-as) o un enlazador (avr-ld). Destacable es la utilidad avr-objcopy, que permite extraer datos de un archivo de código objeto y convertirlo a otro formato, tal como el formato *Intel Hex* que será el utilizado con la herramienta de programación; como también lo es avr-objdump, que permite desensamblar y obtener información de este tipo de archivos.

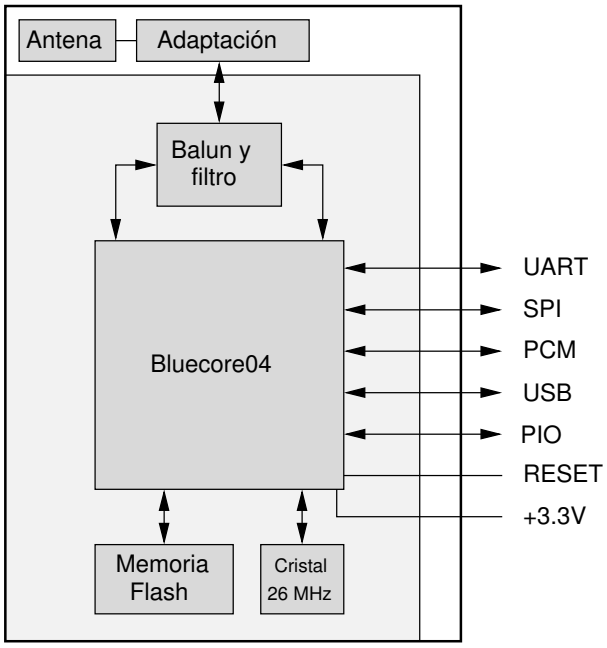

Figura 4.2: Diagrama de bloques del WT12

AVRDUDE

Es una utilidad [4] que permite la grabación, lectura y manipulación de los diversos tipos de memoria de los que consta un microcontrolador AVR. Entre sus características se encuentran las siguientes:

- Interfaz por linea de comandos para todas las tareas de lectura o grabación (incluyendo la manipulación de *fuses*), lo que facilita su inclusión en archivos tipo *Makefile*.
- Permite examinar y modificar de forma interactiva diversas regiones de memoria en el llamado *terminal-mode*.
- Soporta diversos sistemas operativos, aunque en este caso sólo ha sido utilizado en Linux.
- Es capaz de trabajar con un amplio rango de programadores, desde los basados en puerto paralelo sin circuitería adicional a otros más complejos como el AVRISP mkII utilizado para este proyecto.
- Soporta como entrada diversos formatos, como *Intel Hex* o Motorola *S-Record*.

GCC

Se trata de una colección de compiladores desarrollada en el ámbito del proyecto GNU [5]. Es el componente clave de la cadena de herramientas GNU, y uno de los compiladores de más importancia en la actualidad, hasta el punto de que su soporte se considera esencial para el éxito de algunas arquitecturas por parte de algunos fabricantes.

Aunque soporta diversos lenguages, en este proyecto sólo se utilizó como compilador de C, generando código para x86 (en el caso del NCAP) y Atmel AVR (en el caso del TIM), que son dos de las arquitecturas que soporta.

AVR Libc

Proporciona una librería en C para su uso con los microcontroladores AVR de Atmel [6]. Esto incluye archivos de cabecera (.h) que contienen, entre otras cosas, las direcciones de puertos y nombres de registros, la librería de coma flotante o el código de inicialización. Sin ella habría que referirse a cada uno de los registros a través de su dirección, además de escribir este código de inicialización.

Conviene aclarar en este punto como se relacionan todas estas herramientas mencionadas hasta ahora. El proceso comienza generalmente en el compilador, en este caso avr-gcc. Esto convierte el código en lenguaje C a un conjunto de archivos en lenguaje en ensamblador. Estos archivos son equivalentes a los que se tendrían si se hubiera escrito el programa en ensamblador, algo que normalmente se desea evitar.

Este conjunto de archivos en lenguaje ensamblador son convertidos a código objeto, papel que realiza el paquete binutils a través de la herramienta avr-as. Estos archivos con código objeto podrían ser ejecutados por un microcontrolador AVR, el problema es que no hay un único archivo con el que se pueda programar. Hay tantos archivos con código objeto como de código fuente de entrada, y faltan cosas como el código de inicialización. El enlazador (avr-ld) toma todo estos archivos y resuelve referencias cruzadas entre ellos para generar un sólo código objeto. También toma modulos de la librería (AVR Libc) y los añade a este código. Normalmente esto código objeto tiene formato ELF, que gracias a la utilidad avr-objcopy se puede convertir a formato Intel Hex. El compilador, ensamblador, enlazador y la librería forman en núcleo del proceso. Estos programas están altamente interconectados, y son desarrollados conjuntamente en cierta forma, aunque se trate de proyectos separados.

El último paso consiste en el uso de avrdude para programar el microcontrolador con el código de nuestro programa. También existen otras herramientas como avr-gdb y simulavrxx para depurar el programa y emular el dispositivo hardware con el que se trabaja, aunque no han sido utilizados en este caso. En la Figura 4.3 se muestra un diagrama completo del proceso. Finalmente, en la tabla 4.1 se hace un resumen de todas estas herramientas y alguna más no comentada.

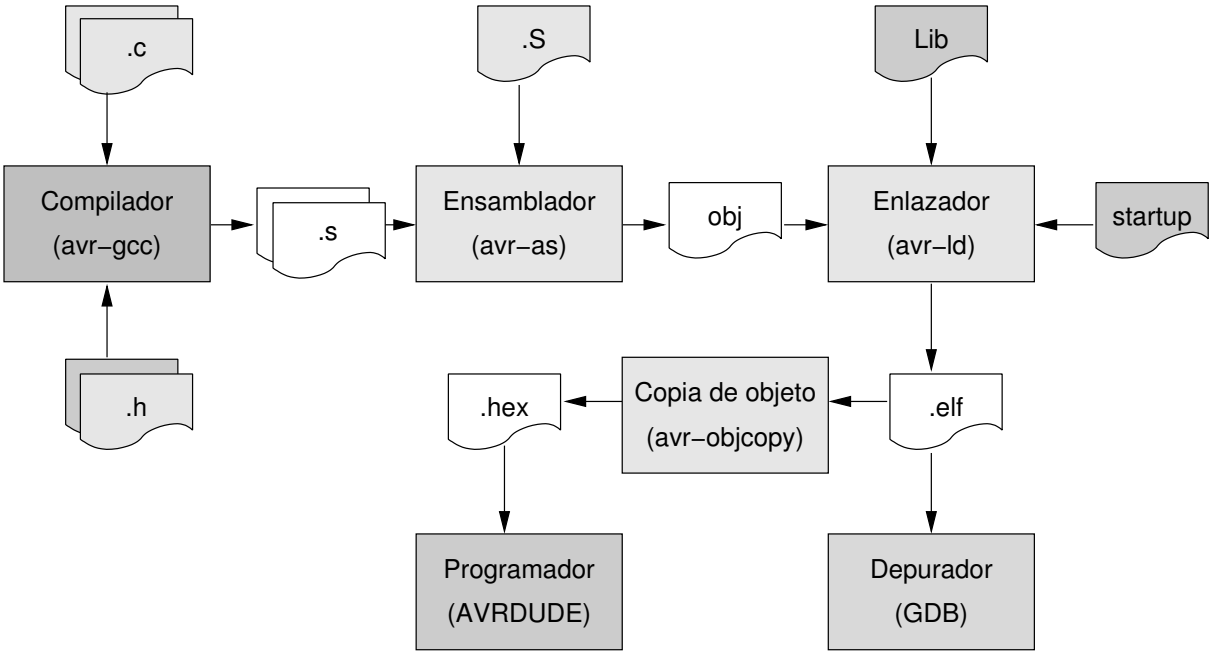

Figura 4.3: Herramientas utilizadas para la programación del ATmega8

Herramienta	Descripción	Proyecto
avr-gcc	Compilador cruzado de C para AVR	gcc
avr-as	Ensamblador para AVR	binutils
avr-ld	Enlazador para enlazar código objeto AVR	binutils
avr-size	Muestra la cantidad de memoria FLASH o RAM ocupada	binutils
avr-nm	Extrae la tabla de símbolos de un archivo de código objeto	binutils
avr-objdump	Muestra información de diversos formatos de código objeto de AVR	binutils
avr-objcopy	Extrae información de archivos objeto y convierte a otros formatos	binutils
avr-libc	Cabeceras y librerías en tiempo de ejecución para AVR	avr-libc
avrdude	Permite programar el microcontrolador	avrdude
avr-gdb	Depurador para AVR	gdb

Tabla 4.1: Resumen de aplicaciones para la programación AVR

BlueZ

Esta librería [7] se ha utilizado para la comunicación Bluetooth en el NCAP. Se trata de una implementación de la pila Blueooth para Linux. Posee varias características interesantes:

- implementación completamente modular
- soporte para múltiples dispositivos Bluetooth
- abstracción real del hardware
- interfaz tipo *socket* para todas las capas
- soporte de seguridad a nivel de dispositivo y de servicio

Entre sus módulos se pueden citar los siguientes:

- subsistema Bluetooth para el núcleo
- capas L2CAP y SCO (audio) para el núcleo
- implementaciones RFCOMM, BNEP, CMTP y HIDP
- librerías y *demonios* Bluetooth y SDP
- utilidades de configuración
- herramientas para el análisis de protocolos

4.3 TIM

La implementación del wTIM llevada a cabo no resultó demasiada compleja, además se intentó soportar todas las características y funciones que aparecen como requeridas en el estándar IEEE 1451.0, incluyendo algunas de las opcionales. La única limitación a la hora de realizar esta implementación vino impuesta por la cantidad de memoria de programa disponible en el microcontrolador ATmega8 (8kB), que afecta especialmente al tamaño de los TEDS disponibles para lectura. Debido al soporte de varios canales en el TIM, este número de TEDS puede ser importante. Por otro lado, debido a la

limitación de memoria tipo EEPROM (512 bytes), tampoco se pudo permitir la escritura de todos los TEDS, sólamente la de uno de ellos, y con un límite de tamaño bastante estricto.

Para el control de la información de TIM y todos los canales se utilizó una matriz de estructuras como la siguiente:

```
struct web_server {
        int socket;
        int flags;
        struct gethandler *gethandler;
        struct web_client *client;
};

struct channel
{
        uint32_t condition;
        uint32_t event;
        uint32_t mask;  // mascara utilizada en solicitudes de servicio
        uint8_t state;  //estado
        uint8_t samplemode; //modo de muestreo
        uint8_t buffered;
        uint8_t txmode;
        uint8_t eods;
        uint8_t ahaltmode;
        uint8_t etr;
        uint8_t trigger;
        uint8_t type;
        uint8_t olddata;
        uint16_t groups [5]; //grupos a los que pertenece el canal
        uint8_t data [10]; //10 bytes para los datos de cada canal
        uint16_t (*pfread) (uint8_t*);
        void (*pfwrite) (uint8_t*, uint16_t);
        void (*pftrigger) (void);
} channels [N_CHANNELS+1];
```

El canal 0 hace referencia al TIM, mientras que los demás valores se corresponden con los diferentes canales implementados. En cuanto a los diferentes miembros de la estructura, se pueden hacer las siguientes aclaraciones:

- Las variables `condition`, `event` y `mask` se utilizan para el control de errores y del estado del dispositivo tal como aparece en el estándar.
- `samplemode` se utiliza para indicar el modo de muestreo en el que está configurado un determinado canal. Se implementaron los modos de muestreo con activación por *trigger* e inmediato, aunque para este proyecto sólo el inmediato era necesario.
- `buffered`, `txmode`, `eods`, `ahaltmode` y `etr` se utilizan para indicar la configuración de varios parámetros. Éstos son los siguientes: activación del modo *buffered*, modo de transmisión, modo de actuación tras el procesado de un conjunto de datos por parte de un actuador, acción a seguir en caso de parada de un actuador (terminar con un conjunto de datos o mantenerlo en memoria) y configuración de un sensor de eventos. Obviamente no se necesitaron todos estos parámetros en este caso, pero se soporta la lectura de ellos a través de comandos 1451.

- `trigger` indica si un sensor o actuador está listo para capturar o procesar datos y puede esperar la recepción de un disparo (ver página 43 de la norma IEEE 1451.0).
- El tipo de transductor (sensor, actuador o sensor de eventos) se indica en la variable `type`.
- `olddata` se utiliza para comprobar que un mismo conjunto de datos no se haya leído dos veces. Si es así, se indica el error correspondiente en las variables de estado.
- `groups` se utiliza para indicar los grupos a los que pertenece un transductor. Por razones de capacidad de memoria, se ha limitado a 5.
- Los datos de un transductor son almacenados en `buffer` (nuevamente limitados por la memoria disponible).
- `pfread`, `pwrite` y `pftrigger` son punteros a funciones que son llamadas en caso de llegada de un comando lectura (sensor), escritura (transductor) o *trigger* (ambos casos). Estas funciones deben encargarse de dejar los datos en la matriz `buffer` o bien leerlos de ella. En muchos casos, implementar un nuevo TIM 1451 se podría reducir a sustituir estas funciones por otras, manteniendo el intercambio de datos a través de este buffer.

4.4 NCAP

En el caso del NCAP se realizó una implementación conjunta con un servidor HTTP para poder implementar parte del API HTTP definido en el IEEE 1451.0. Aunque usualmente sólo se utiliza un servidor, es posible ejecutar varios servidores HTTP simultáneos con este mismo NCAP. Para cada servidor se tiene una estructura como la siguiente:

```
struct web_server {
        int socket;
        int flags;
        struct gethandler *gethandler;
        struct web_client *client;
};
```

Como se puede ver, se mantiene el socket asociado al servidor, una lista de clientes y una lista de *handlers*, cada uno de ellos asociada a una determinada URL. Por ejemplo, a la hora de crear un manejador para la URL de lectura de un sensor basta con una llamada a la función `web_server_addhandler` de la siguiente forma:

```
web_server_addhandler (&server, "/ReadData", func_readdata1,func_readdata2, 0);
```

Esta función espera como parámetros, además del servidor al que se quiere asociar y una URL, un par de funciones que serán las encargadas de procesar la petición. La primera de ellas (`func_readdata1`) es la encargada de lanzar una petición al TIM, mientras que la segunda func_readdata2 se encarga de procesar los datos recibidos del TIM y generar una respuesta de tipo texto. Estas dos funciones deben recibir como único parámetro un puntero a la estructura que maneja un cliente y un puntero a una estructura de solicitud a un TIM respectivamente. Su valor de retorno debe ser 0 en caso de que no se desee servir aún la petición de un cliente (ya que se espera la respuesta de un TIM) y destruir su estructura asociada. En caso contrario, se retorna el número de bytes que se han añadido a la respuesta. Por tanto, los dos casos más normales son los siguientes:

- Si se trata de una petición que no debe esperar ninguna respuesta del TIM, la primera función retornaría el número de bytes que ha generado como respuesta. Como puntero a la segunda función se pasaría `NULL` a `web_server_addhandler`.
- Si se debe esperar una respuesta del TIM, la primera función debe retornar 0 (aunque también podría añadir datos a la respuesta si se desea), y es la segunda la que retorna el número de bytes que ha añadido a la respuesta.

La estructura utilizada para manejar las peticiones HTTP tiene la siguiente forma:

```
struct web_client {
        int socket;
        struct sockaddr_in sa;
        unsigned int salen;
        char *rbuf;
        char *sbuf;
        int rbufsize;
        int sbufsize;
        int newdata_try;
        int writedsize;
        unsigned long contentlength;
        unsigned long headersize;
        int wheadersize;
        int writelength;
        int range;
        char *HTTPdirective;
        unsigned char stat;
        char *buffer;
        int bufsize;
        int c;

        char *request;
        char *method;
        char *QueryString;
        char *PostData;
        struct memrequest *mem;
        struct _Header *HeaderList;
        struct _Query *QueryList;
        struct _Post *PostList;
        struct _MultiPart *MultiPartList;

        struct web_server *server;
        struct web_client *next;
        struct web_client *prev;

        struct gethandler *gh;
};
```

Cuando llega una petición HTTP se crea una estructura de este tipo y se le añade un puntero al *handler* que se encargará de procesarla (si no existe la URL se genera un error 404). En caso de que exista la URL, se llama a la primera función del *handler*. La estructura posee miembros relacionados

con el envío y recepción de datos, y otros relacionados con el tratamiento de las cabeceras o distintos parámetros de un GET o un POST.

Por otro lado, cada uno de los TIMs tiene asociada esta estructura:

```
struct tim {
        uint16_t id;
        char *rbuf;
        unsigned int rsize;
        char state;
        struct trequest *treq;
        int fd;
        float timeout;
        timer_id timerid;
        uint8_t *metateds;
        uint16_t metatedssize;
        struct tedscache *tc;
        struct tim *next;
        struct tim *prev;
};
```

Mantiene miembros para el almacenamiento de informaciones tales como el identificador asociado al TIM, un tiempo de espera para las respuestas de los comandos 1451 (o tiempo de procesado antes de poder enviar el siguiente comando en el caso de que no se espere ninguna respuesta), una lista de las solicitudes pendientes de mandar al TIM y una cache de TEDS tal como recoge el estándar IEEE 1451.0. Las solicitudes se recogen en la siguiente estructura:

```
struct trequest {
        char state;
        char *rbuf;
        unsigned int rsize;
        char *sbuf;
        unsigned int ssize;
        struct web_client *client;
        void (*func) (struct tim *ptim);
        struct trequest *next;
};
```

Básicamente contiene información [1] los datos que se quieren envir al TIM y los recibidos, un puntero al cliente HTTP para el que se ha generado la solicitud y un puntero a una función. Para generar una de estas solicitudes se hace una llamada a la función `add_request`, que recibe esta información como parámetros:

```
add_request (struct tim *ptim, char *buff, unsigned int size, struct web_client *
client, void (*func) (struct tim *ptim));
```

Una vez procesada la solicitud (por recepción de una respuesta o bien por cumplirse el tiempo de espera) se llamará a una determinada función, dependiendo de los siguientes casos:

- Si la función `add_request` se llamó con un `NULL` en el parámetro `client`, se llamará a la función a la que apunta el miembro `func` de la estructura `trequest`. Esto es útil para peticiones a un TIM que no se producen por la petición de un cliente HTTP; es decir, para el intercambio de información interna entre el TIM y el NCAP.

- En el caso de que `client` no sea `NULL` se buscará la función a la que se debe llamar en su *handler* asociado (la segunda de las dos que contiene cada *handler*, que es la que debería estar destinada a procesar la respuesta del TIM).

- Si tanto `func` como `client` son `NULL`, no se generará ninguna llamada. Este caso se da cuando se quiere enviar información al TIM sin preocuparse de ninguna respuesta.

Otro aspecto a destacar es que todas las comunicaciones son no bloqueantes, y se gestionan a través del módulo `demux.c`, el cual mantiene una lista de *descriptores de archivo* y sus funciones *callback* asociadas. Se encarga de comprobar si hay datos disponibles en uno de estos descriptores, llamando a la función correspondiente en caso de que sea así. Nótese que esta comprobación se hace a través de la llamada al sistema `select`, manteniéndose por tanto el uso de la CPU al mínimo.

Finalmente se realizó un pequeño interfaz WEB para poder acceder al NCAP desde cualquier navegador WEB. Se puede acceder a él a través de la URL http://host:1451/Interface. En la figura 4.4 se muestra su aspecto.

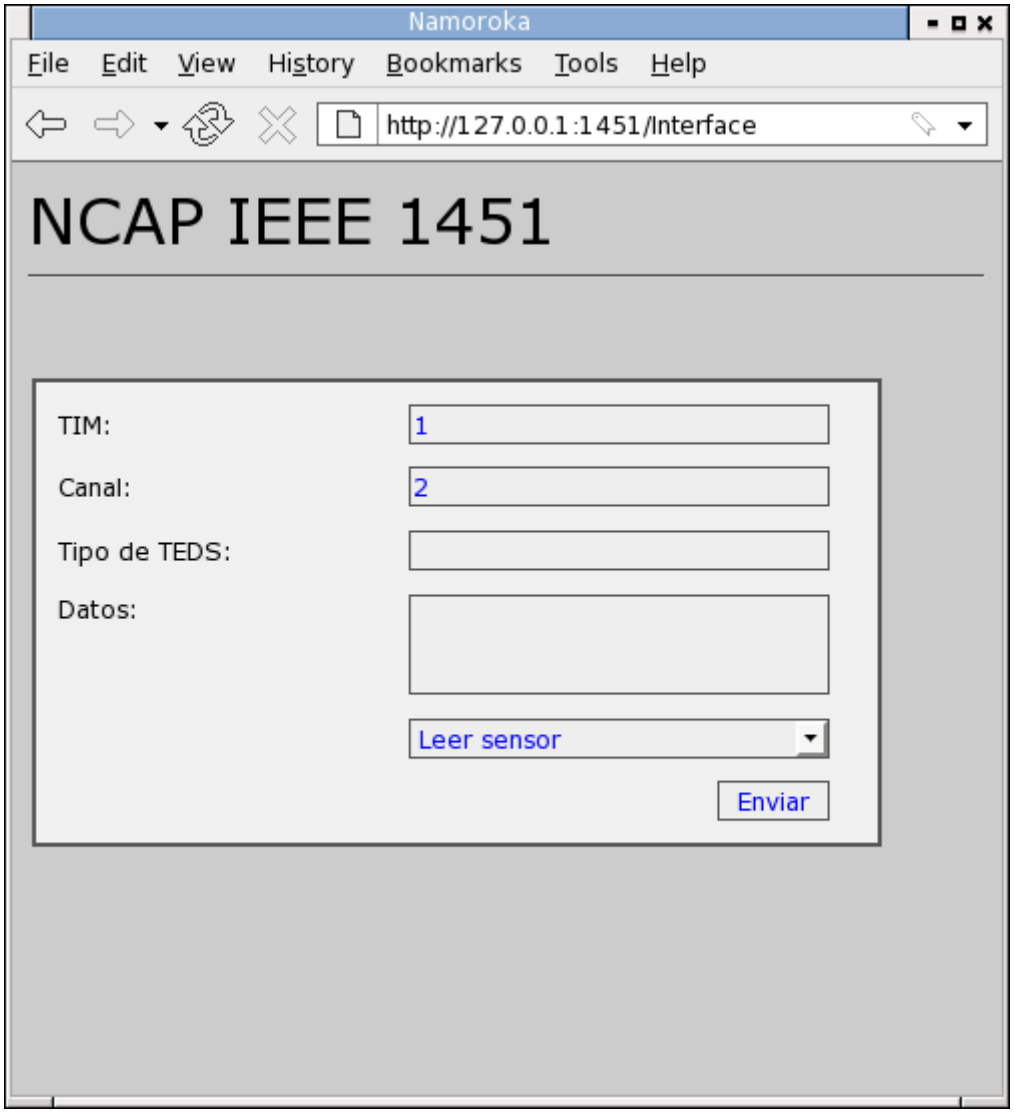

Figura 4.4: Interfaz WEB realizado para el NCAP

Puesto que todas las comunicaciones con el hardware se basan en descriptores de archivo, es de esperar que permitir accesos a TIMs basados en otras capas físicas apenas suponga modificaciones al código, salvo para el descubrimiento de los dispositivos. En este caso, para el descubrimiento de los dispositivos Bluetooth se utilizó un simple *Inquiry* seguido de una conexión SDP para identificar transductores IEEE 1451. Puesto que no parece haber ningún UUID asignado para este servicio, se escogió uno de entre todos los que hay disponibles para poder realizar pruebas.

Referencias

[1] http://www.atmel.com.

[2] http://www.bluegiga.com.

[3] http://www.gnu.org/software/binutils/.

[4] http://www.bsdhome.com/avrdude/.

[5] http://gcc.gnu.org.

[6] http://www.nongnu.org/avr-libc/.

[7] http://www.bluez.org.