Capítulo 5
Implementación de la Pila TCP/IP

Para que un microcontrolador se pueda conectar a una red basada en protocolos TCP/IP, se deben implementar las funciones descritas en el capítulo 4. En el capítulo 3 se explicó la implementación física del sistema, el cual cuenta con los componentes necesarios para que pueda ser conectado físicamente a una red y transferir información a través de ella. En este capítulo se describe la implementación del software que permite que la comunicación se realice.

La estructura general de cualquier proyecto para aplicaciones web basado en este software se divide en tres bloques principales:

1) El software que maneja al controlador de red del microcontrolador
2) La pila de protocolos
3) La aplicación específica.

Cada uno de los bloques tiene funciones de recepción y envío de datos. La pila de protocolos TCP/IP se compone de varios archivos que interactúan entre sí para realizar todo el proceso de comunicación, estos archivos corresponden a un protocolo de la pila TCP/IP como se presentan en la Tabla 5.1.

Tabla 5.1 Listado de archivos de la pila TCP/IP

Protocolo	Archivos
IP	ip.c ip.h
ARP	arp.c arp.h
ICMP	icmp.c icmp.h
TCP	tcp.c tcp.h
UDP	udp.c udp.h
HTTP	http.c http.h
Controlador de red	ne64api.h timers.h address.h ne64config.h MC9S12NE64.h

El archivo de cabecera MC9S12NE64.h contiene todas las definiciones del microcontrolador, incluidos nombres de registros y de puertos y se encuentra incluido en el compilador. El archivo ne64api.h contiene las definiciones de las funciones del controlador de red. Dichas funciones las proporciona el fabricante del microcontrolador y en esta sección se describe su funcionamiento.

La pila de protocolos se diseña en este capítulo y su implementación se realiza en lenguaje C con el compilador Metrowerks CodeWarrior IDE para la familia 9S12. Todos los módulos de la pila de protocolos se dividen en dos partes:

1) Un archivo de cabecera (terminación *.h) que incluye definiciones de valores empleados, como las estructuras de datos usadas en cada protocolo o valores constantes configurados por el usuario, como tiempos de espera.
2) El archivo con el código fuente (terminación *.c) que implementa el protocolo en cuestión.

La programación de una aplicación de usuario se analiza en el capítulo 7 donde se hace uso de las funciones de la pila desarrollada en este capítulo para crear aplicaciones de control vía web con el sistema de desarrollo presentado en el capítulo 3.

En la Figura 5.1 se muestra un diagrama general de la comunicación entre los protocolos de la pila TCP/IP en el cual cada protocolo tiene funciones de recepción y envío de datos. La comunicación comienza cuando se recibe una petición por parte de un dispositivo remoto. Esta petición la recibe el controlador de red de

donde se pasa a uno de los protocolos de la capa de red, dependiendo del código incluido en el paquete. Este proceso se repite en cada capa hasta que los datos llegan a uno de los protocolos o funciones de aplicación, en donde se genera una respuesta que se va regresando a los niveles inferiores hasta regresar al controlador de red en donde se realiza el envío de la respuesta. El diagrama mostrado representa el flujo normal de información, aunque puede truncarse en cualquier punto si los datos recibidos contienen errores.

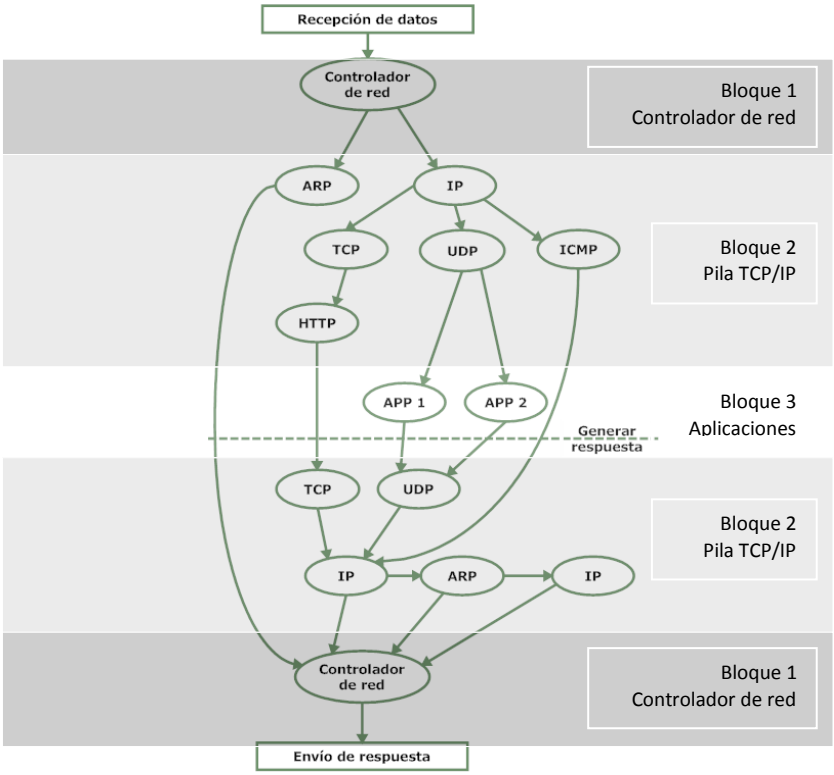


Figura 5.1 Flujo de datos de una petición realizada al microcontrolador

5.1 Controlador de red

La programación del software del controlador de red depende del hardware empleado. Para el caso del controlador Ethernet incluido en el microcontrolador MC9S12NE64, *Freescale* proporciona el conjunto de funciones que configuran, inicializan y manejan al controlador de red. El desarrollo de los niveles de software superiores se basa en la programación proporcionada por el fabricante del microcontrolador.

5.1.1 Configuraciones

El archivo *ne64config.h* proporciona una interfaz de usuario que permite configurar opciones del controlador de red para determinar su modo de operación mediante variables simbólicas. Las configuraciones realizadas se escriben en los registros apropiados del microcontrolador cuando éste se inicializa. En la Tabla 5.2 se muestran las configuraciones del controlador de red. Para más detalles sobre el modo de operación del controlador de red del microcontrolador consultar el tema 3.2.4.

Tabla 5.2 Configuraciones del controlador de red

Configuración	Macro	Valores posibles	
Autonegociación	`AUTO_NEG`	0	Deshabilitado
		1	Habilitado
Anuncio de habilidades en	`HALF100` `FULL100`	0	No anuncia el modo de transmisión indicado

modo de autonegociación	HALF10 FULL10	1	Anuncia el modo de transmisión indicado
Modo de transmisión cuando no se usa autonegociación	SPEED100	0	Velocidad de 100 Mbps
		1	Velocidad de 10 Mbps
	FULL_DUPLEX	0	Modo Full Dúplex
		1	Modo Half Dúplex
Buffers de memoria	BUFMAP	0	Tamaño de buffers 128
		1	Tamaño de buffers 256
		2	Tamaño de buffers 512
		3	Tamaño de buffers 1024
		4	Tamaño de buffers 1536
Filtrado de paquetes por dirección de destino y control de recepción de flujo	XFLOWC	0	Habilita control de flujo
		1	Deshabilita control de flujo
	BRODC_REJ	0	Acepta tramas broadcast
		1	Rechaza tramas broadcast
	CON_MULTIC	0	Acepta tramas multicast
		1	Revisa validez de tramas multicast
	PROM_MODE	0	Habilita filtro por dirección
		1	Deshabilita filtro por dirección, acepta todas las tramas
Filtro por tipo de paquete (ethertype)	ETYPE_PET ETYPE_EMW ETYPE_IPV6 ETYPE_ARP ETYPE_IPV4 ETYPE_IEEE	0	Rechaza paquetes con el tipo indicado
		1	Acepta paquetes con el tipo indicado
	ETYPE_ALL	0	Habilita filtro por tipo de paquete
		1	Deshabilita filtro por tipo de paquete, acepta todos los tipos

El módulo EMAC del microcontrolador filtra automáticamente los paquetes recibidos con base al contenido de los campos de *dirección de destino* y *tipo/longitud* del encabezado Ethernet. El algoritmo de filtrado por dirección de destino que ejecuta el controlador de red se muestra en la Figura 5.2 y en la Figura 5.3 se muestra el algoritmo de filtrado por tipo de paquete. El diagrama del módulo físico Ethernet y de Control de Acceso al Medio se puede consultar en el capítulo 3 en el tema 3.2.4.

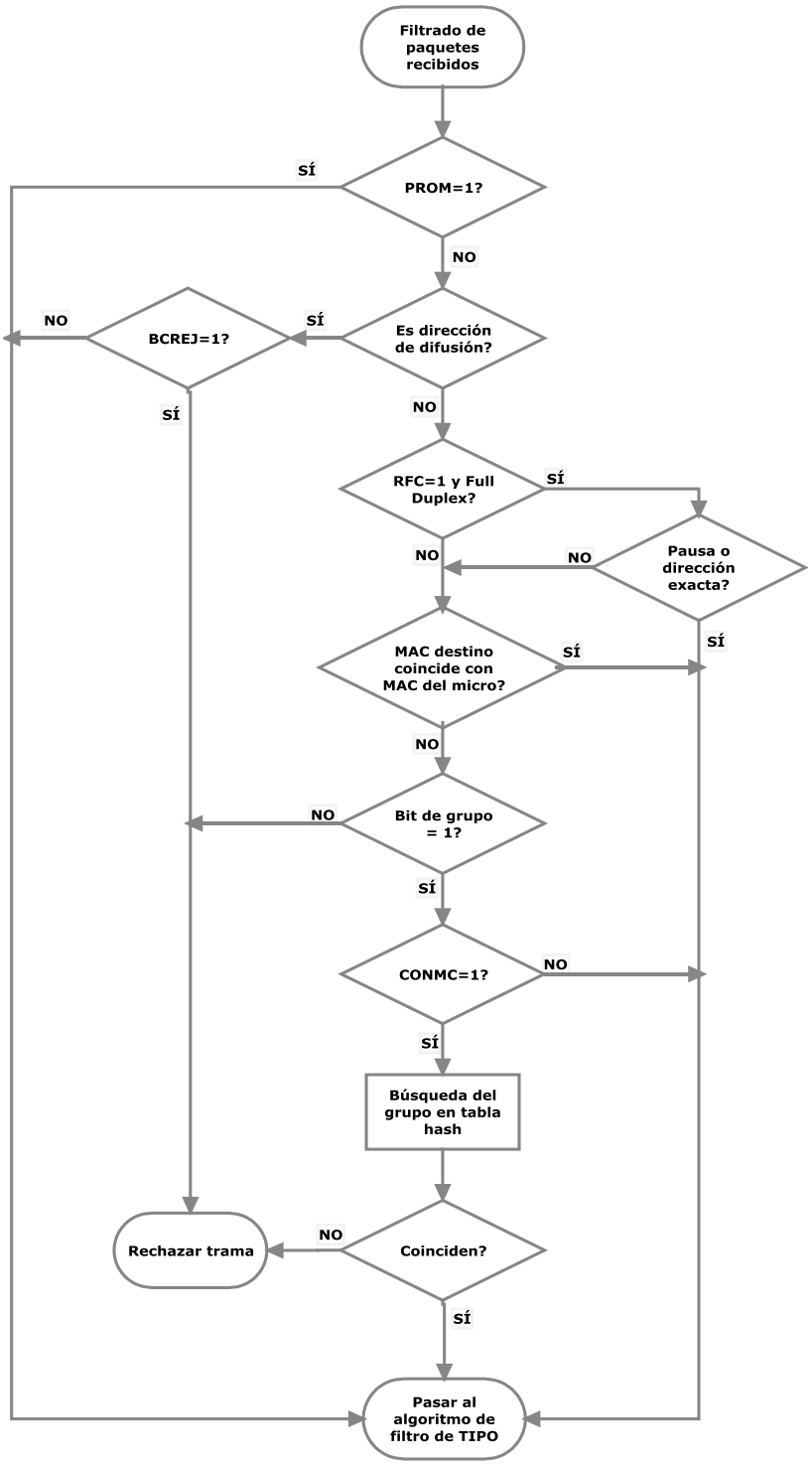


Figura 5.2 Algoritmo de filtro de paquetes por dirección destino [5e]

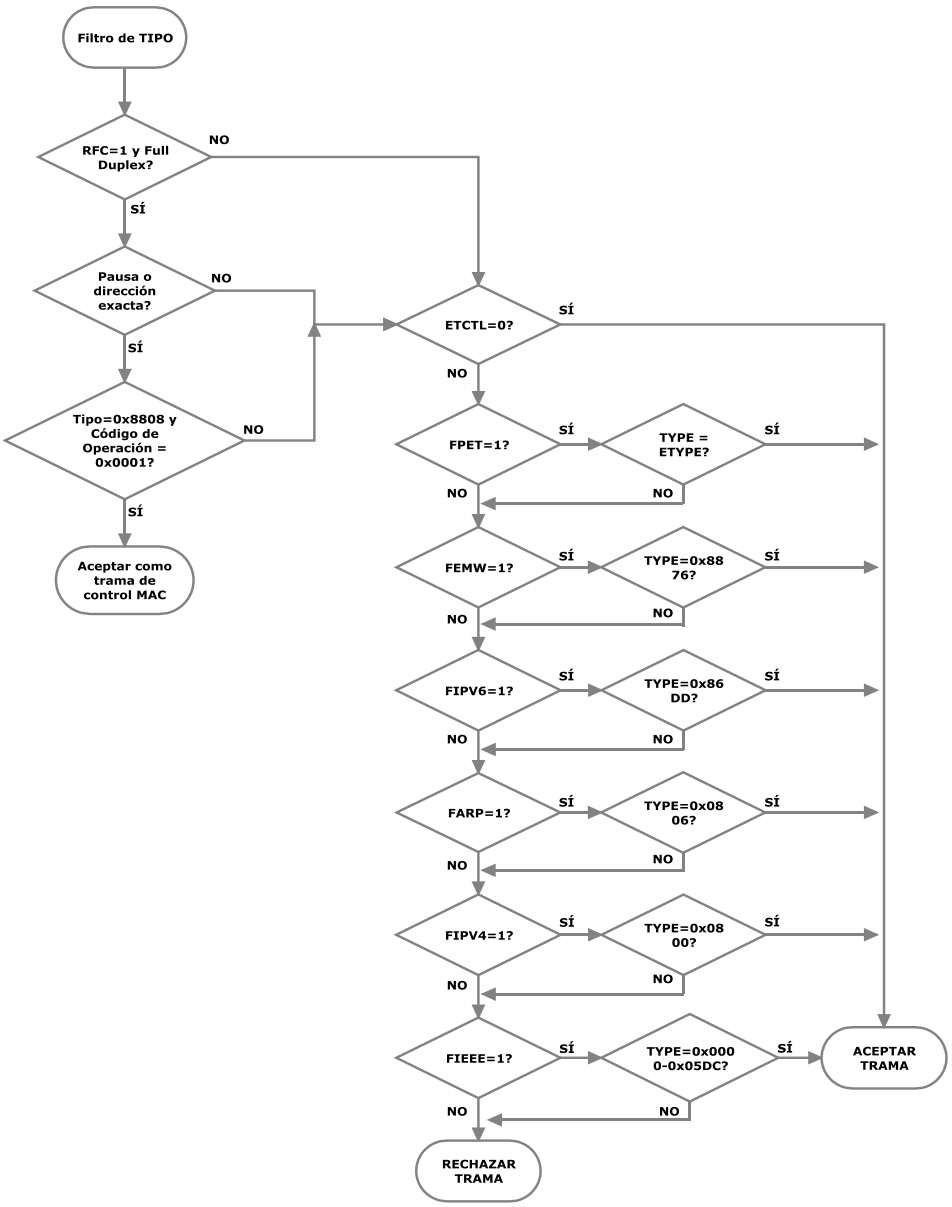


Figura 5.3 Algoritmo de filtro de paquetes por tipo/longitud [5e]

5.1.2 Direcciones

En el archivo *address.c* es posible configurar los valores de dirección de red, dirección física así como máscara de subred y dirección del Gateway de la red. El valor de la dirección MAC que se configura en este archivo se transfiere al registro MACAD de 48 bits del módulo EMAC en el momento de la inicialización. El contenido del archivo se muestra a continuación.

```
const tU08 hard_addr[6] = {0x01, 0x23, 0x45, 0x56, 0x78, 0x9a};

const tU08 prot_addr [4] = { 192, 168,   2,   3 };
const tU08 netw_mask [4] = { 255, 255, 255,   0 };
const tU08 dfgw_addr [4] = { 192, 168,   2,   1 };
const tU08 brcs_addr [4] = { 192, 168,   2, 255 };
```

En la primera variable se escribe la dirección física, o dirección MAC, del controlador de red. En las siguientes variables se configura, respectivamente, la dirección IP, la máscara de subred, la dirección del gateway default de la red y la dirección broadcast del nivel de red. Todos estos datos se presentan como arreglos de números enteros de 8 bits, de manera que cada dato del arreglo corresponde a uno de los números enteros que se encuentran separados por puntos en las direcciones de red.

5.1.3 Implementación del software del controlador de red

En el archivo *ne64driver.c* se implementa el código fuente del controlador de red. En este archivo podemos encontrar la declaración de las variables que permiten manipular los espacios de memoria destinados a los buffers de recepción y transmisión. Estos espacios de memoria se localizan al inicio de la memoria RAM y tienen los tamaños definidos por la macro BUFMAP en el archivo de configuración. Para escribir o leer datos de estos buffers se declaran tres arreglos localizados al inicio de la RAM. Parte de la memoria que queda por debajo de estos arreglos se destina a memoria RAM de propósito general.

En la Figura 5.4 se ilustra la localización de las variables anteriores. Supongamos que la memoria RAM comienza en la dirección 0x2000 y que la configuración de BUFMAP corresponde al valor de 2, lo cual indica que el tamaño de cada uno de los buffers es de 512 bytes. Las macros EMAC_RX_SZ y EMAC_TX_SZ corresponden al valor del tamaño de los buffers, en este caso 512. Con lo anterior tenemos que el buffer de recepción A ocupa las localidades de memoria que van de la 0x2000 a la 0x21FF, el buffer de recepción B de la 0x2200 a la 0x23FF y el buffer de transmisión de la 0x2400 a la 0x25FF. La memoria RAM comienza a partir de la dirección 0x2600. Para leer el primer dato del buffer de recepción A se lee el valor de emacFIFOa[1], el segundo dato se encuentra en emacFIFOa[2] y así sucesivamente. De manera similar para escribir un dato al inicio del buffer de transmisión se escribe un dato en emacFIFOtx[1], para escribir en la segunda posición se escribe en emacFIFOtx[2], etcétera.

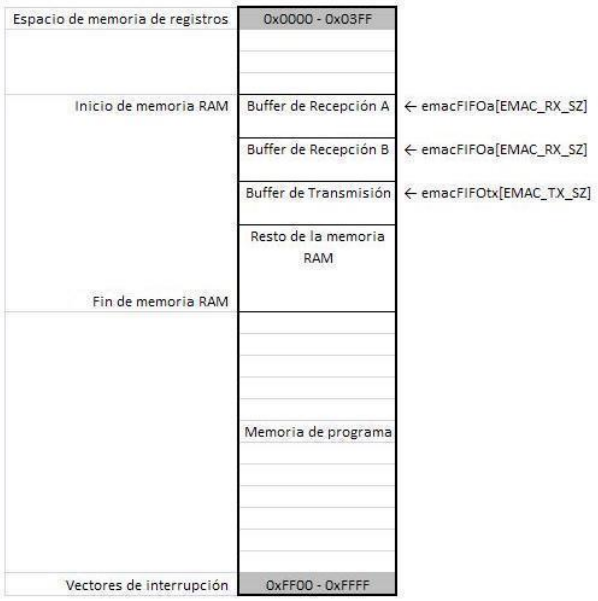


Figura 5.4 Distribución del espacio de memoria del microcontrolador y localización de los buffers de recepción y transmisión

En la Tabla 5.3 se presentan las acciones que realiza el controlador de red y el nombre de la función que implementa cada acción.

Tabla 5.3 Funciones del controlador de red

Acción que realiza	Nombre de la función
inicialización	EtherInit().
Transmisión de datos	EtherStartFrameTransmission()
Recepción de datos	NE64Receive()

A continuación se describen las funciones presentadas en la Tabla 5.3 y su forma de uso.

void **EtherInit**(void). Inicializa el controlador de red y ajusta el modo de operación de acuerdo a las configuraciones seleccionadas en el archivo ne64config.h. No recibe ni regresa valores. Se debe llamar al inicio de la aplicación desde la función principal como parte de la inicialización general del sistema y de la aplicación.

Para llamar a esta función: `EtherInit();`

void **EtherStartFrameTransmission**(tU16 datalen). Envía los datos almacenados en el buffer de transmisión, recibe como parámetro la longitud de los datos a enviar, expresada como un número entero de 16 bits sin signo. Al llamar a esta función los datos ya deben estar almacenados en el buffer. No tiene valor de retorno. Esta función se llama desde el protocolo superior que requiere enviar datos, recordemos que los protocolos que se encuentran sobre Ethernet son IP y ARP.

Para llamar a esta función: `EtherStartFrameTransmission(100);`

En el ejemplo el controlador transmite 100 datos que se encuentran almacenados en la memoria de transmisión.

interrupt void **emac_rx_b_a_c_isr**(void), interrupt void **emac_rx_b_b_c_isr**(void). Son rutinas de interrupción que manejan la recepción de datos en los buffer A y B respectivamente. Ambas funciones hacen una llamada a la función NE64Receive.

void **NE64Receive**(void *PktBuffer, UINT16 len, UINT16 flags). Es la función utilizada por las rutinas de interrupción de recepción. Recibe como parámetros un apuntador a la localidad de memoria en donde se almacenan los datos, que puede ser el buffer A o el buffer B. El segundo parámetro corresponde a la longitud de los datos que se recibieron, para el buffer A este dato se lee del registro *Puntero de Fin de Frame de Recepción A* (RXAEFP) y para el buffer B del registro *Puntero de Fin de Frame de Recepción B* (RXBEFP). El último parámetro es una bandera de 16 bits, que indica cual de los dos buffers está llamando a la función.

Para llamar a esta función desde la rutina de interrupción del buffer A:
```
NE64Receive(emacFIFOa, RXAEFP, IEVENT&(IEVENT_RXACIF_MASK));
```

Para llamar a esta función desde la rutina de interrupción del buffer B:
```
NE64Receive(emacFIFOb, RXBEFP, IEVENT&(IEVENT_RXBCIF_MASK));
```

Esta función bloquea las interrupciones en el buffer que recibe datos para evitar que una nueva recepción borre los datos almacenados antes de que sean procesados. Al término del procesamiento debe llamarse a otra función que libera al buffer para que pueda seguir recibiendo datos.

Después de bloquear las interrupciones la función guarda la información de la trama en una estructura de datos que almacena los valores de las cabeceras Ethernet y una referencia a la ubicación de los datos dentro de la trama que se encuentra almacenada en alguno de los buffers de recepción. Esta variable la usan los protocolos de los niveles superiores para realizar sus funciones. La estructura se define como un nuevo tipo de dato.

```
typedef struct TEthernetFrame
{
      UINT8        destination[ETH_ADDRS_LEN];
      UINT8        source[ETH_ADDRS_LEN];
      UINT16       frame_size;
      UINT16       protocol;
      UINT16       buf_index;
} TEthernetFrame;
```

El tipo de dato TEthernetFrame guarda información sobre una trama Ethernet, la cual corresponde a la dirección MAC de destino, la dirección MAC de origen y el protocolo superior. La variable *frame_size* almacena la longitud de la trama y la variable *buf_index* guarda un número entero que indica la posición del comienzo de los datos en el buffer de almacenamiento.

5.1.4 API

En el archivo *ne64api.c* se implementa una Interfaz de Programación de Aplicaciones (API) que usan los protocolos de nivel superior. Su archivo de cabecera *ne64api.h* contiene la declaración de las funciones de la API, que se pueden dividir en funciones de escritura a buffer y transmisión de datos y funciones de recepción de datos y lectura de buffer. El archivo de cabecera se debe incluir al inicio de cualquier archivo que haga uso de las funciones del API del controlador de red.

Funciones de escritura y transmisión de datos

Para transmitir una trama primero se debe inicializar el buffer de transmisión, después se escribe el encabezado y por último los datos. Para la escritura de los datos existen tres funciones. En la Tabla 5.4 se presentan las acciones de escritura, transmisión y el nombre de la función que implementa cada acción.

Tabla 5.4 Funciones de escritura y transmisión de datos

Acción que realiza	Nombre de la función
Inicialización del buffer de transmisión	NE64InitializeTransmissionBuffer()
Escribir encabezado al buffer	NE64WriteEthernetHeaderToTxBuffer()
Escribir un byte al buffer	NE64WriteByte()
Escribir una palabra (2 bytes)al buffer	NE64WriteWord()
Escribir bytes al buffer	NE64WriteBytes()

A continuación se describen las funciones presentadas en la Tabla 5.4.

void **NE64InitializeTransmissionBuffer**(void). Se debe llamar a esta función antes de escribir cualquier dato en la memoria de transmisión. Aquí se inicializa un apuntador que guarda la dirección del inicio del buffer de transmisión. Este apuntador es una variable global que utilizan las funciones de escritura para almacenar datos en el buffer, después de hacerlo, incrementan el valor del contador para que la siguiente vez que se use una función de escritura los datos se almacenen a partir de la siguiente dirección libre.

void **NE64WriteEthernetHeaderToTxBuffer**(TEthernetFrame *frame). Esta función recibe como parámetro un apuntador a una estructura *TEthernetFrame* que almacena los datos del encabezado Ethernet, mismos que escribe en el buffer de transmisión.

void **NE64WriteByte**(tU08 dat). Escribe un dato de 8 bits en la localidad apuntada por la variable global e incrementa su valor una unidad.

void **NE64WriteWord**(tU16 dat). Escribe un dato de 16 bits en la localidad apuntada por la variable global e incrementa su valor dos unidades.

void **NE64WriteBytes**(tU08 *buf, tU16 len). Escribe una cierta cantidad de datos, indicada por el parámetro *len*. Los datos se toman de una localidad especificada por el apuntador *buf*. El apuntador global se incrementa en un número igual a la cantidad de bytes escritos en el buffer de transmisión.

Una vez que los datos han sido acomodados en el buffer de transmisión, se procede a su envío llamando a la siguiente función:

void **NE64StartFrameTransmission**(tU16 len). Realiza una llamada a la función *EtherStartFrameTransmission*, descrita anteriormente.

Funciones de lectura y recepción de datos

Al recibir una trama, primero se verifica que alguno de los buffers de recepción contenga datos válidos, cuando sea el caso, se inicializa una lectura sobre el buffer correspondiente, se leen los datos y se finaliza la recepción liberando el buffer. Existen tres funciones diferentes para leer datos de los buffers. En la Tabla 5.5 se presentan las acciones de recepción y lectura de datos y el nombre de la función que implementa cada acción.

Tabla 5.5 Funciones de recepción y lectura de datos

Acción que realiza	Nombre de la función y sus parámetros
Verificación de recepción de datos válidos	NE64ValidFrameReception()
Inicialización del desplazamiento para leer el buffer	NE64InitializeOffsetToReadRxBuffer()
Leer un byte del buffer	NE64ReadByte()
Leer una palabra (2 bytes) del buffer	NE64ReadWord()
Leer bytes del buffer	NE64ReadBytes()
Liberar el buffer	NE64FreeReceiveBuffer()

A continuación se describen las funciones presentadas en la Tabla 5.5.

UINT16 **NE64ValidFrameReception**(void). Esta función se debe llamar para saber si hay datos listos para ser procesados en alguno de los buffers de recepción, en caso de que los haya, regresa un valor mayor a cero y se inicializa la lectura. Si el buffer aun no termina su actividad de recepción, o no contiene datos almacenados, regresa el valor de cero.

void **NE64InitializeOffsetToReadRxBuffer**(tU16 pos). En el caso de la recepción también se maneja un apuntador global que indica la posición del buffer de recepción de donde se leen los datos. Antes de llamar a cualquier función de lectura se debe llamar a esta función indicando la posición del buffer que se quiere leer. Cada vez que alguna función lea un dato de la memoria de recepción, el apuntador se incrementa a la siguiente localidad que aún no ha sido leída.

UINT8 **NE64ReadByte**(void). Lee y regresa un byte de la localidad apuntada por la variable global y la incrementa una unidad.

UINT16 **NE64ReadWord**(void). Lee y regresa una palabra de 16 bits de la localidad apuntada por la variable global y la incrementa dos unidades.

void **NE64ReadBytes**(tU08 *buf, tU16 len). Lee una cierta cantidad de datos, indicada por el parámetro *len*, y los almacena en la localidad de memoria especificada por el apuntador *buf*. El apuntador global se incrementa en un número igual a la cantidad de bytes leídos del buffer de recepción.

void **NE64FreeReceiveBuffer**(void). Cuando los datos almacenados en alguno de los buffers de recepción ya se han leído, procesado o almacenado en alguna variable, el buffer de recepción debe ser liberado, desbloqueando las interrupciones, para que puedan almacenarse nuevos datos de entrada. Con esta función se libera al buffer de recepción.

5.2 Uso de temporizadores

El manejo de los tiempos es una característica que comparten varios protocolos de la pila, esto genera la necesidad de usar más de un contador. Un método comúnmente empleado consiste en configurar un contador del microcontrolador para que genere una interrupción cada cierto intervalo de tiempo. Por otro lado se crean variables que almacenan números enteros que se incrementan o decrementan en la rutina de servicio de interrupción periódica, lo que les permite acumular tiempos.

Para generar el periodo de interrupción se usa el módulo del microcontrolador llamado *Interrupción de Tiempo Real*, o *RTI*. Este módulo utiliza la frecuencia del oscilador de entrada, OSCCLK, y la divide entre un preescalador para generar una frecuencia base para las interrupciones. El valor del divisor se configura en el registro RTICTL, mostrado en la Figura 5.5, de acuerdo a los valores de la Tabla 5.6.

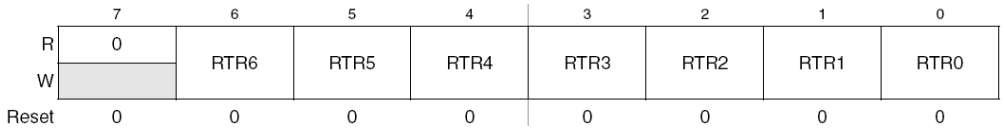


Figura 5.5 Registro de Control de Interrupción de Tiempo Real, RTICTL

Los bits de este registro se dividen en dos grupos: los primeros cuatro definen un valor base del preescalador y los últimos tres le dan un rango más amplio al multiplicarlo por una potencia de 2. Al seleccionar alguna de las configuraciones de la tabla la frecuencia del oscilador de entrada se divide entre el valor resultante (únicamente para el módulo RTI) y se genera una interrupción por cada pulso de la señal resultante.

En la Figura 5.6 se muestran los periodos resultantes (en milisegundos) para cada configuración de preescalador utilizando una frecuencia de entrada de 25 MHz. La posición de cada uno corresponde a los preescaladores de la Tabla 5.6.

Como se puede observar, el tiempo de interrupción más grande que se puede lograr está dado en aproximadamente 42 ms, correspondiente al preescalador $16x2^{16}$. Observando los valores obtenidos se selecciona el tiempo más aproximado a 1 ms (sombreado en la Figura 5.6), para que se utilicen contadores de tiempo en milisegundos y su manejo sea práctico.

En la Tabla 5.6 se muestran resaltados los valores del preescalador que generan interrupciones cada milisegundo aproximadamente y se muestran dos configuraciones posibles del registro RTICTL:

- $0100\ 0010_2$ ($0x28 = 66_{10}$)
- $0011\ 0101_2$ ($0x35 = 53_{10}$)

Tabla 5.6 Valores del divisor de frecuencia para la Interrupción de tiempo Real

RTR[0:3]	RTR[4:6]							
	000 (OFF)	001 (2^{10})	010 (2^{11})	**011** (2^{12})	**100** (2^{13})	101 (2^{14})	110 (2^{15})	111 (2^{16})
0000	OFF	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001	OFF	$2x2^{10}$	$2x2^{11}$	$2x2^{12}$	$2x2^{13}$	$2x2^{14}$	$2x2^{15}$	$2x2^{16}$
0010	OFF	$3x2^{10}$	$3x2^{11}$	$3x2^{12}$	$3x2^{13}$	$3x2^{14}$	$3x2^{15}$	$3x2^{16}$
0011	OFF	$4x2^{10}$	$4x2^{11}$	$4x2^{12}$	$4x2^{13}$	$4x2^{14}$	$4x2^{15}$	$4x2^{16}$
0100	OFF	$5x2^{10}$	$5x2^{11}$	$5x2^{12}$	$5x2^{13}$	$5x2^{14}$	$5x2^{15}$	$5x2^{16}$
0101	OFF	$6x2^{10}$	$6x2^{11}$	$6x2^{12}$	$6x2^{13}$	$6x2^{14}$	$6x2^{15}$	$6x2^{16}$
0110	OFF	$7x2^{10}$	$7x2^{11}$	$7x2^{12}$	$7x2^{13}$	$7x2^{14}$	$7x2^{15}$	$7x2^{16}$
0111	OFF	$8x2^{10}$	$8x2^{11}$	$8x2^{12}$	$8x2^{13}$	$8x2^{14}$	$8x2^{15}$	$8x2^{16}$
1000	OFF	$9x2^{10}$	$8x2^{11}$	$8x2^{12}$	$8x2^{13}$	$8x2^{14}$	$8x2^{15}$	$8x2^{16}$
1001	OFF	$10x2^{10}$	$10x2^{11}$	$10x2^{12}$	$10x2^{13}$	$10x2^{14}$	$10x2^{15}$	$10x2^{16}$
1010	OFF	$11x2^{10}$	$11x2^{11}$	$11x2^{12}$	$11x2^{13}$	$11x2^{14}$	$11x2^{15}$	$11x2^{16}$
1011	OFF	$12x2^{10}$	$12x2^{11}$	$12x2^{12}$	$12x2^{13}$	$12x2^{14}$	$12x2^{15}$	$12x2^{16}$
1100	OFF	$13x2^{10}$	$13x2^{11}$	$13x2^{12}$	$13x2^{13}$	$13x2^{14}$	$13x2^{15}$	$13x2^{16}$
1101	OFF	$14x2^{10}$	$14x2^{11}$	$14x2^{12}$	$14x2^{13}$	$14x2^{14}$	$14x2^{15}$	$14x2^{16}$
1110	OFF	$15x2^{10}$	$15x2^{11}$	$15x2^{12}$	$15x2^{13}$	$15x2^{14}$	$15x2^{15}$	$15x2^{16}$
1111	OFF	$16x2^{10}$	$16x2^{11}$	$16x2^{12}$	$16x2^{13}$	$16x2^{14}$	$16x2^{15}$	$16x2^{16}$

RTICTL =

0.0410	0.0819	0.1638	0.3277	0.6554	1.3107	2.6214
0.0819	0.1638	0.3277	0.6554	1.3107	2.6214	5.2429
0.1229	0.2458	0.4915	0.9830	1.9661	3.9322	7.8643
0.1638	0.3277	0.6554	1.3107	2.6214	5.2429	10.4858
0.2048	0.4096	0.8192	1.6384	3.2768	6.5536	13.1072
0.2458	0.4915	0.9830	1.9661	3.9322	7.8643	15.7286
0.2867	0.5734	1.1469	2.2938	4.5875	9.1750	18.3501
0.3277	0.6554	1.3107	2.6214	5.2429	10.4858	20.9715
0.3686	0.7373	1.4746	2.9491	5.8982	11.7965	23.5930
0.4096	0.8192	1.6384	3.2768	6.5536	13.1072	26.2144
0.4506	0.9011	1.8022	3.6045	7.2090	14.4179	28.8358
0.4915	0.9830	1.9661	3.9322	7.8643	15.7286	31.4573
0.5325	1.0650	2.1299	4.2598	8.5197	17.0394	34.0787
0.5734	1.1469	2.2938	4.5875	9.1750	18.3501	36.7002
0.6144	1.2288	2.4576	4.9152	9.8304	19.6608	39.3216
0.6554	1.3107	2.6214	5.2429	10.4858	20.9715	41.9430

Figura 5.6 Periodos de interrupción en ms para una entrada de 25 MHz

Las variables que se usan como contadores o temporizadores deben tener ciertas características, como son:

Tipo. Indica si un contador es ascendente o descendente. Los contadores ascendentes pueden servir para medir el tiempo que transcurre entre un evento y otro, se crean al inicio de un intervalo de tiempo y se lee el valor que registran al concluir dicho intervalo. Los contadores descendentes sirven para definir intervalos de tiempo, se crean y se inicializan con un valor y al llegar a cero indican que se ha cumplido dicho intervalo.

Estado. El estado de una variable tipo contador indica si el contador está libre y disponible para usarse o si ya se encuentra en uso.

Valor. El valor del contador indica la cantidad de veces que se ha cumplido la interrupción, por lo tanto contiene el valor del tiempo, mismo que depende del periodo de interrupción. Para este caso el valor de un contador indica la cantidad de milisegundos que almacena la variable, si es ascendente indica cuántos milisegundos han transcurrido desde que se creó, si es descendente indica cuántos milisegundos faltan para que se venza el tiempo inicial.

Se define un conjunto de funciones que opera sobre un conjunto limitado de contadores. En la Tabla 5.7 se muestra el nombre elegido, la sintaxis y definición de cada función.

Tabla 5.7 Funciones de manejo de temporizadores

Función	Descripción	Ejemplo
timers_init();	Inicializa todos los contadores. El tipo se pone como *descendente*, el estado como *libre* y el valor inicial *cero*.	timers_init();

create_timer();	Busca un timer disponible y cambia su estado a *ocupado*. Lo inicializa con el tipo y valor indicados en los dos argumentos que recibe. Tiene un tercer argumento que permite indicar si el valor se envía como segundos o milisegundos. Regresa un indicador numérico del contador que se obtiene (en caso de encontrar alguno libre), este número sirve para realizar funciones sobre el contador. En caso de que no haya contadores libres al momento de llamar a esta función se reinicializa el código, ya que el manejo de los tiempos es crucial.	timer1=create_timer(UP, 0, MS);
		timer2=create_timer(DOWN, 3, SEC);
		timer3=create_timer(DOWN, 250, MS);

reset_timer_value();	Permite reescribir un valor a un contador que se encuentra en uso. Recibe como parámetro el número de contador, el nuevo valor que debe contener y sus unidades.	reset_timer_value(timer1, 0, MS);
		reset_timer_value(timer2, 8, SEC);
		reset_timer_value(2, 300, MS);

read_timer_ms();	Recibe como parámetro el identificador de un contador y regresa su valor expresado en milisegundos.	time_left = read_timer_ms(timer1);
read_timer_seconds();	Recibe como parámetro el identificador de un contador y regresa su valor expresado en segundos.	time_left = read_timer_seconds(timer2);
get_timer_type();	Regresa el tipo del contador identificado con el parámetro que recibe.	get_timer_type(timer1); get_timer_type(2);
toggle_timer_type();	Cambia el tipo de un contador.	toggle_timer_type(timer1); toggle_timer_type(2);
free_timer	Libera el contador indicado por el parámetro que recibe, pone su estado como *libre* para que pueda ser usado nuevamente.	free_timer(timer1); free_timer(2);

En la rutina de interrupción RTI se examina cada uno de los contadores que se encuentran en uso y dependiendo de su tipo su valor se incrementa o decrementa una unidad (correspondiente a un milisegundo).

5.3 Protocolo de Resolución de Direcciones (ARP)

La implementación de este protocolo debe considerar las acciones mostradas en la Tabla 5.8, las cuales están basadas en la descripción presentada en el tema 4.5 y corresponden a las recomendaciones del RFC 826 [20e]. Esta tabla también muestra el nombre que se eligió para las funciones que implementan dichas acciones y un diagrama que muestra la interacción entre las funciones del protocolo.

Para almacenar los mensajes ARP se define la siguiente estructura, correspondiente al formato de mensajes ilustrado en la Figura 4.5:

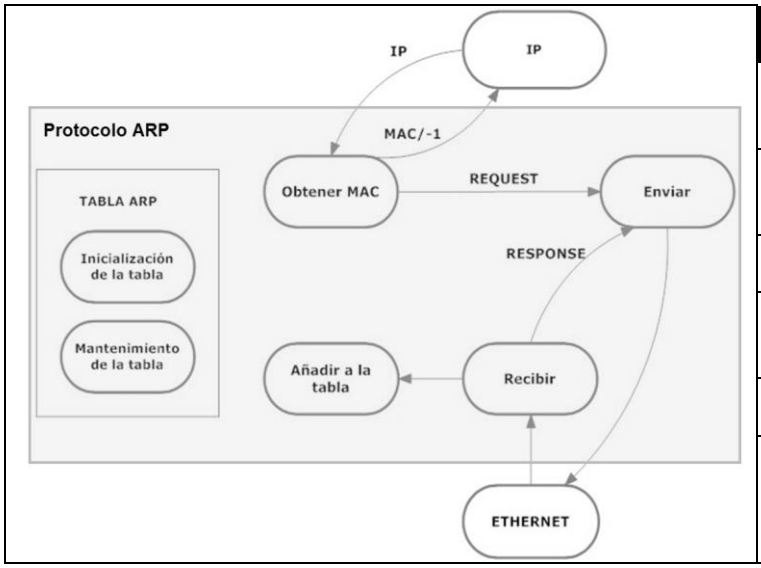
```
struct arp_message{
  unsigned int  HWtype;
  unsigned int  protocol;
  unsigned char HWdirlen;
  unsigned char protdirlen;
  unsigned int  opcode;
  unsigned char MACaddSource[6];
  unsigned char IPaddSource[4];
  unsigned char MACaddDest[6];
  unsigned char IPaddDest[4];
};
```

La tabla de direcciones ARP se almacena usando la siguiente estructura, de la cual se declara un arreglo que contiene todas las entradas de la tabla:

```
struct arp_table{
  unsigned char status;
  unsigned char type;
  unsigned char ttl;
  unsigned char retries;
  unsigned char MACaddr[6];
  unsigned char IPaddr[4];
} ;
```

El *status* indica el estado de la entrada, puede ser libre, pendiente (en espera de la respuesta) o resuelta. El tipo indica si se trata de una entrada estática o una dinámica. La variable *ttl* especifica el tiempo de vida de la entrada, esta variable está controlada por un temporizador, si una solicitud no es respondida y el temporizador vence, el tiempo de vida se renueva mientras queden intentos de envío de solicitud. Cuando el temporizador vence y no quedan más intentos de reenvío la entrada se libera. Una vez que la entrada ha sido resuelta, la variable *ttl* almacena el tiempo de validez de dicha entrada, al expirar el tiempo de vida la entrada se libera si es de tipo dinámico, o se renueva si es de tipo estático. La variable *retries* guarda una memoria de la cantidad de intentos de que dispone una entrada para enviar una solicitud y esperar una respuesta. Una vez que la solicitud ha sido atendida exitosamente, las variables *MACaddr* e *IPaddr* guardan la relación entre ambas direcciones del dispositivo remoto.

Tabla 5.8 Funciones del protocolo ARP



Acción que realiza	Nombre de la función
Inicialización de valores (tabla de entradas ARP)	arp_init()
Búsqueda de datos en la tabla (Obtener MAC)	arp_get_mac()
Envío de solicitudes y respuestas	arp_send()
Recepción de solicitudes y respuestas	arp_receive()
Añadir direcciones a la tabla	arp_add()
Rutina de mantenimiento de la tabla	arp_manage()

5.3.1 Inicialización

La función de inicialización del módulo ARP se encarga de ajustar el estado de todas las entradas de la tabla a *LIBRE* (o *FREE*) y generar el contador que maneja al módulo. Este contador se crea de tipo descendente (*DOWN*) y se inicializa con un valor de 1 segundo, periodo en el cual se lleva a cabo la rutina de mantenimiento de la tabla ARP. Esta función se debe llamar desde la rutina principal como parte de la inicialización general del sistema, antes de ejecutar cualquier función de la pila.

```
void arp_init();
```

5.3.2 Procesamiento de salida de datos

Cuando un protocolo de nivel de red, en este caso IP, requiere enviar paquetes hacia otro dispositivo remoto del cual únicamente conoce su dirección de red, hace uso de la función de ARP denominada **arp_get_mac()**, que recibe como parámetro la dirección IP, con la intención de conocer la dirección MAC asociada, como se puede observar en la Figura 4.3. Esta función busca dentro de la tabla ARP la correspondiente dirección física, si la encuentra la regresa a la función IP que la llamó y en caso contrario envía una solicitud regresando un entero negativo. El diagrama de flujo que implementa el algoritmo anterior se muestra en la Figura 5.7. Para llamar a esta función desde el nivel de red:

```
ip[]={192,168,2,3};
tmp = arp_get_mac(ip);
```

La variable tmp almacena el valor de retorno de la función arp_get_mac(), que puede ser la misma MAC buscada o un entero negativo que indica que la MAC no se encuentra en la tabla.

La búsqueda de la MAC se realiza recorriendo cada una de las entradas de la tabla y verificando su estado. Cuando se encuentra una entrada en estado pendiente, se revisa si la IP corresponde a la que se está buscando, en caso de serlo se regresa un –1 que indica que la MAC aún no se encuentra (se espera por la respuesta). Cuando se encuentra una entrada resuelta se examina la dirección IP y si corresponde a la buscada se regresa la dirección MAC correspondiente (se encontró). En la Figura 5.7 se muestra la rutina de búsqueda sobre la tabla de direcciones ARP.

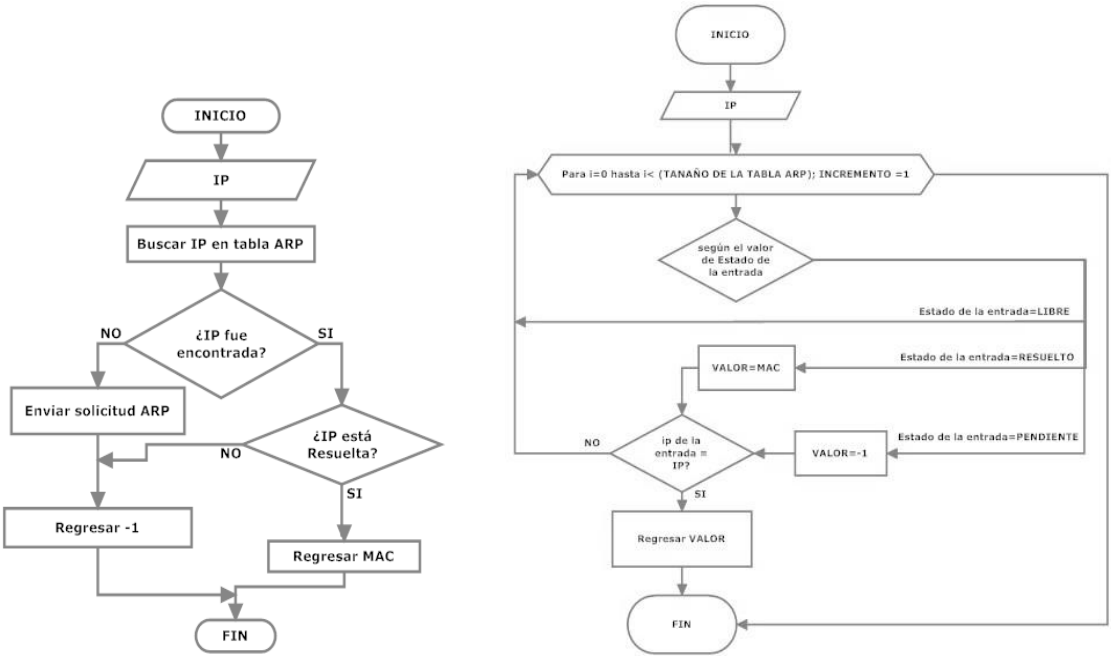


Figura 5.7 Izquierda, algoritmo de la función de obtención de MAC llamada por el protocolo de red. Derecha, procedimiento de búsqueda de dirección MAC a partir de dirección IP

Cuando la dirección MAC no se encuentra y no hay ninguna entrada en la tabla que contenga el valor de la IP del destino, se envía una solicitud ARP mediante la función **arp_send()**. Esta función recibe tres parámetros y sirve tanto para enviar solicitudes como para enviar respuestas, lo cual se indica mediante su primer parámetro. El segundo parámetro corresponde a la dirección IP del destino. El tercer parámetro corresponde a la dirección MAC del destino para el caso de la respuesta y para el caso de la solicitud es cero. No tiene valor de retorno. arp_send() es una función utilizada dentro del módulo ARP por otras funciones de este módulo, para el caso del envío de solicitudes se llama de la siguiente manera:

```
arp_send(REQUEST, ip, 0);
```

En la Figura 5.8 se muestra el diagrama de flujo de la función de envío de ARP. Primero se fija la dirección de destino, ya sea la dirección broadcast cuando se trata de una petición, o la dirección MAC indicada al llamar a la función en caso de una respuesta. La construcción de la trama Ethernet y del paquete ARP se realizan utilizando variables declaradas como estructuras para cada protocolo (struct TEthernetFrame y struct arp_message). Una vez llenas estas variables con los valores adecuados se utilizan las funciones del controlador para inicializar el buffer de transmisión, escribir el encabezado Ethernet, escribir los datos de la variable arp_message e iniciar la transmisión (funciones descritas en el tema 5.1.4).

5.3.3 Procesamiento de entrada de datos

Cuando se recibe un paquete destinado al protocolo ARP se llama a la función **arp_receive()** desde el software del controlador de red. Esta función recibe como parámetro un número entero que indica la posición de los datos ARP en el buffer de recepción (por ejemplo 14, ya que los datos almacenados al inicio del buffer corresponden al encabezado Ethernet – datos 0 al 13). No tiene valor de retorno. Para llamar a esta función desde el nivel inferior:

```
arp_receive(address_buff);
```

En la Figura 5.9 se muestra el diagrama de flujo del algoritmo de recepción de paquetes ARP. La inicialización del buffer de lectura corresponde a la función del controlador descrita en el tema 5.1.4. Después de llamar a esta función se utilizan las funciones de lectura del controlador para leer los datos del buffer y se van almacenando en una variable tipo estructura arp_message. Una vez construido el mensaje éste se valida para cerciorarse que contiene la información correcta de acuerdo al protocolo, si se encuentra alguna anomalía se sale de la función sin realizar ningún procesamiento sobre la información.

Cuando el paquete ha sido validado satisfactoriamente, se examina el campo *código de operación* para saber si se trata de una respuesta a una solicitud previamente enviada, o si se trata de una petición por parte de algún dispositivo remoto. En el caso de las peticiones se envía una respuesta llamando a la función **arp_send(REPLY, ip, mac)**, enviando la dirección física y de red del dispositivo al cual se envía la respuesta. Después de enviar la respuesta los datos del emisor se almacenan en la tabla ARP. Si el paquete recibido corresponde a una respuesta, la dirección física y de red del emisor se almacenan en la tabla.

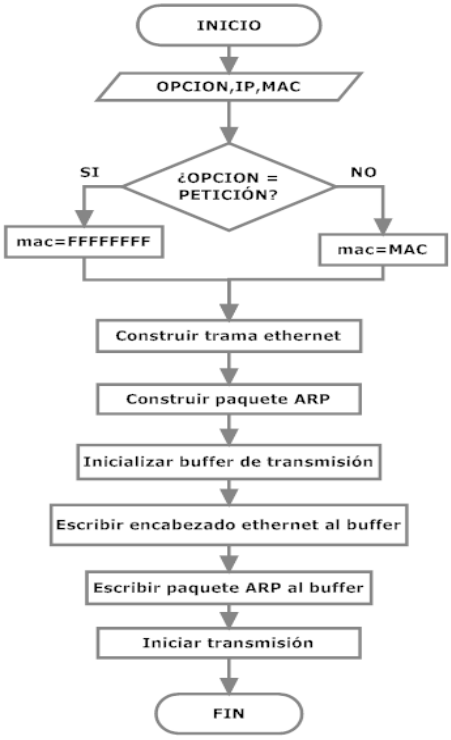


Figura 5.8 Algoritmo de envío de datos ARP

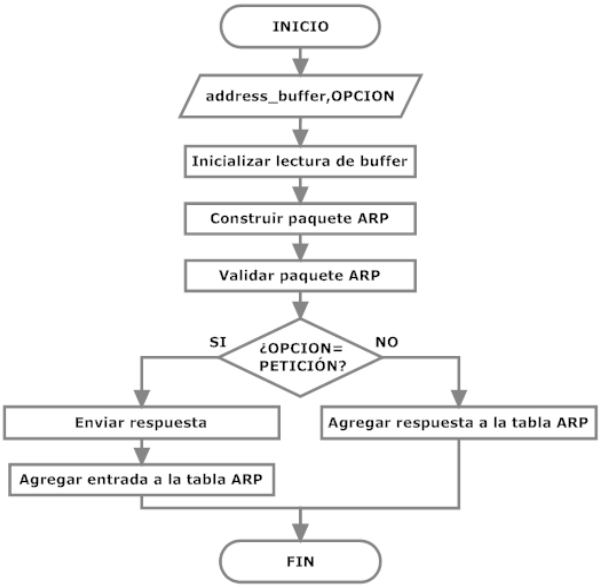


Figura 5.9 Algoritmo de recepción de datos ARP

5.3.4 Mantenimiento de la tabla ARP

La función que se encarga de manejar las entradas almacenadas en la tabla ARP se denomina **arp_manage()**. Esta función deber llamarse de manera periódica desde la rutina principal. No recibe parámetros ni tiene valor de retorno. El algoritmo para esta acción se presenta en la Figura 5.10.

Al entrar a esta función se verifica si el temporizador general ARP ha expirado, recordemos que el valor para este contador se inicializa con un segundo, por lo tanto esta rutina se ejecuta cada segundo. Si aún no transcurre el segundo no se realiza acción alguna. Cuando el contador haya alcanzado el valor de cero, el valor de todos los tiempos de vida de las entradas (valores enteros que expresan segundos) se decrementan, se actualiza el valor del contador ARP para iniciar un nuevo ciclo y se inicia un recorrido por la tabla examinando los estados de cada entrada y el valor de sus contadores. Para entradas pendientes cuyo tiempo de espera por una respuesta ha expirado, si aún quedan intentos se envía una nueva solicitud, de no ser así se libera la entrada. Para entradas resueltas cuyo tiempo de vida ha expirado, se examina su tipo y si corresponde al tipo dinámico la entrada se libera, en caso contrario se renueva, enviando una nueva solicitud.

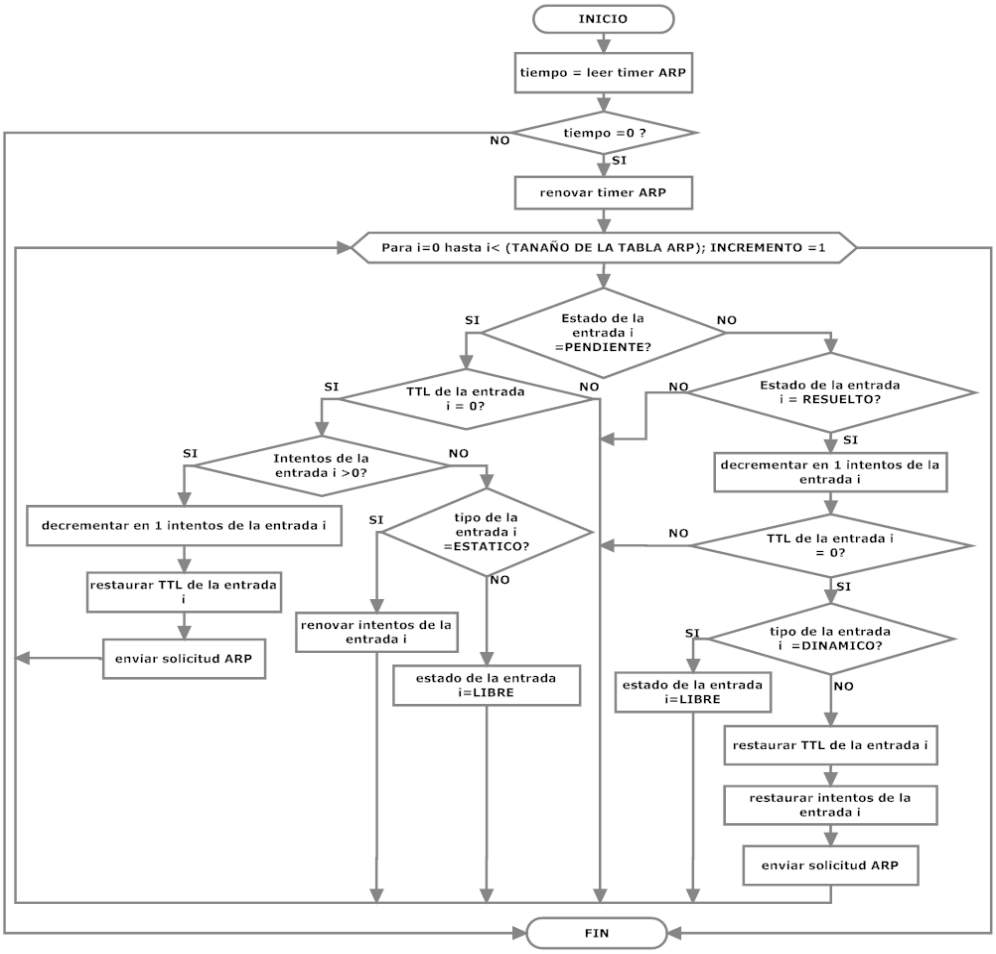


Figura 5.10 Rutina de mantenimiento de las entradas de la tabla ARP

5.4 Protocolo de Internet (IP)

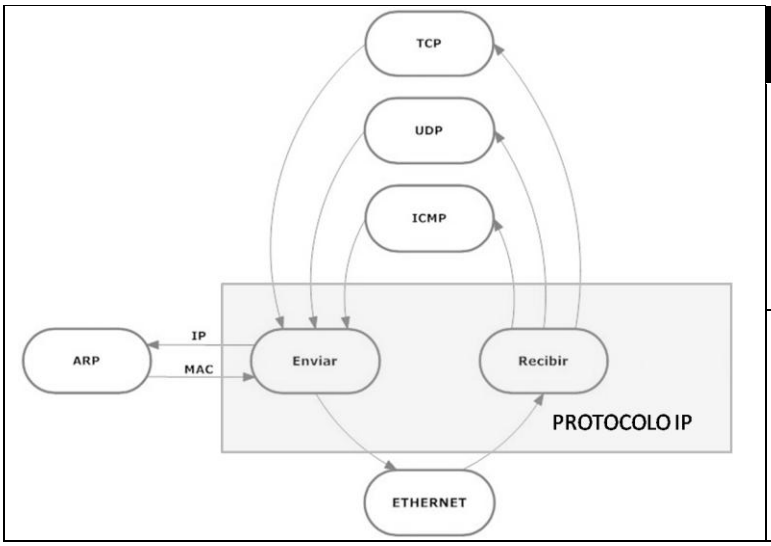
El diseño e implementación del Protocolo de Internet se basa en las recomendaciones del estándar RFC 791 [21e]. La implementación del protocolo para esta pila es sumamente sencilla, ya que no se realizan algoritmos de encaminamiento y no se toman en cuenta paquetes IP fragmentados. El software IP no necesita variables ni estructuras especiales que guarden estados, ni rutinas que deban ser llamadas periódicamente. La implementación se reduce a la entrega y recepción de paquetes entre los protocolos colindantes con IP, es decir, en la transmisión de datos IP recibe paquetes provenientes de los protocolos TCP o UDP de la capa de transporte, o el protocolo ICMP, y los entrega a Ethernet. En la recepción IP recibe paquetes de Ethernet y los entrega a TCP, UDP o ICMP, dependiendo el tipo de paquete recibido. En la Tabla 5.9 se presentan las acciones de recepción y envío que realiza el protocolo IP y el nombre que se eligió para la función que implementa cada acción, así como un diagrama que muestra la interacción de las funciones del protocolo IP con otros protocolos.

La estructura empleada para almacenar datagramas IP corresponde al formato ilustrado en la Figura 4.6 y se define como sigue:

```
struct ip_datagram{
  unsigned char vihl;
  unsigned char tos;
  unsigned int  tlen;
  unsigned int  id;
  unsigned int  flags_offset;
  unsigned char ttl;
  unsigned char protocol;
```

```
  unsigned int  checksum;
  unsigned char IPaddSource[4];
  unsigned char IPaddDest[4];
  unsigned char opt[IP_LEN_BYTE(IP_MAX_OP_LEN)];
  unsigned int  data;
};
```

Tabla 5.9 Funciones del protocolo IP



Acción que realiza	Nombre de la función
Envío de datos	ip_send()
Recepción de datos	ip_receive()

5.4.1 Procesamiento de salida de datos

Cuando un protocolo hace uso de IP para enviar datos, le debe indicar la dirección IP del dispositivo remoto y es tarea de IP determinar a qué dispositivo de la red física enviar el paquete. La función de transmisión de IP se denomina **ip_send()**, se llama desde el protocolo de la capa de transporte y recibe como argumentos la dirección IP remota, un puntero al buffer que contiene los datos, la longitud de datos a ser transferidos y el código del protocolo que envía los datos

El valor de retorno de esta función es un número entero, que es positivo en caso de un envío realizado exitosamente, o negativo en caso contrario. Para llamar a esta función:

```
#define          ICMP_ETYPE     1
#define          UDP_ETYPE      17
#define          TCP_ETYPE      6
ip[]={192,168,2,3};
datos[]="Mensaje enviado a través de IP";
tmp = ip_send(ip,datos,30,UDP_ETYPE);
```

El ejemplo anterior pretende ser ilustrativo, debe quedar claro que los datos enviados a IP deben corresponder al formato del protocolo que los envía. Para el caso del ejemplo, los datos almacenados en el buffer deben contener al inicio los encabezados del protocolo UDP, pero para IP el contenido de los datos no tiene relevancia, sólo necesita recibir la dirección del buffer y la cantidad de datos de ese buffer que debe transferir.

Lo primero que realiza la función ip_send() es explorar la naturaleza de la dirección IP que recibe para determinar hacia dónde debe encaminar el paquete, esto es, si la IP tiene el formato de las direcciones de la subred a la que pertenece el microcontrolador el envío será directo, si IP determina que la dirección está fuera de la red física a la que el microcontrolador está conectado, entonces lo envía al dispositivo de compuerta de red, haciendo una petición de MAC a ARP usando la IP del gateway, pero enviando el paquete con la IP destino y la MAC del gateway.

Una vez determinada la naturaleza de la IP, se busca la dirección física del dispositivo al que se enviará el paquete, haciendo uso de la función arp_get_mac() de ARP. Si el valor regresado es negativo, se sale de la función ip_send(), ya que la dirección física aún no está resuelta. IP no realiza más acciones en estos casos, cuando el protocolo superior determine que los datos que envió no llegaron a su destino, intentará enviarlos de nuevo.

Si se tuvo éxito en la obtención de la dirección física, se procede a formar la trama Ethernet con una variable struct TEthernetFrame y posteriormente el datagrama IP con una variable struct ip_datagram. Una vez llenas estas variables con los valores adecuados se utilizan las funciones del controlador para inicializar el buffer de transmisión, escribir el encabezado Ethernet, escribir los datos de la variable ip_datagram e iniciar la transmisión (funciones descritas en el tema 5.1.4). El algoritmo anterior se ilustra en la Figura 5.11

Para determinar si la IP pertenece a la subred se hace uso de la máscara de subred definida en el archivo ne64address.c (explicado en el tema 5.1.2). Primero se hace una operación AND bit a bit entre la IP examinada y la máscara de subred, y entre la IP del microcontrolador y la máscara de subred, a fin de eliminar de la IP los bits del identificador de host y conservar los del identificador de red, como se explica en el tema 4.6.4. Después se comparan las direcciones resultantes y si son iguales entonces pertenecen a la misma red, en caso contrario la IP remota está en una red externa.

Dentro del proceso de construcción del datagrama IP se calcula la suma de verificación, de acuerdo al algoritmo del complemento a uno de la suma en complemento a uno de los datos del encabezado IP tomados como enteros de 16 bits contiguos (explicada en el tema 4.6.1).

5.4.2 Procesamiento de entrada de datos

La función **ip_receive()** es la encargada de procesar los datos recibidos por el protocolo IP y es llamada desde el software de recepción del controlador de red. Recibe como parámetro un número entero que indica la posición de los datos IP en el buffer de recepción. No tiene valor de retorno. Para llamar a esta función:

```
ip_receive(address_buff);
```

En la Figura 5.12 se ilustra el mecanismo de recepción de datos IP. La inicialización del buffer de lectura corresponde a la función del controlador descrita en el tema 5.1.4. Después de llamar a esta función se utilizan las funciones de lectura del controlador para leer los datos del buffer y se van almacenando en una variable tipo estructura ip_datagram. Una vez construido el datagrama éste se valida para cerciorarse que contiene la información correcta. Si se encuentra alguna anomalía se sale de la función. Dentro de las validaciones realizadas se calcula el checksum para asegurar la integridad de los datos del encabezado IP. Cuando el paquete ha sido correctamente validado, se examina el campo *protocolo* para saber a cuál protocolo enviar el paquete recibido.

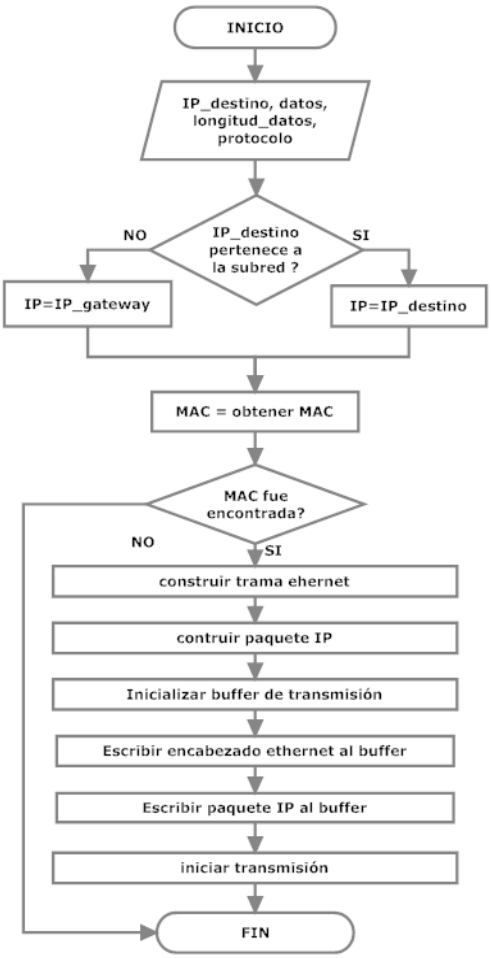


Figura 5.11 Algoritmo de transmisión de datos IP

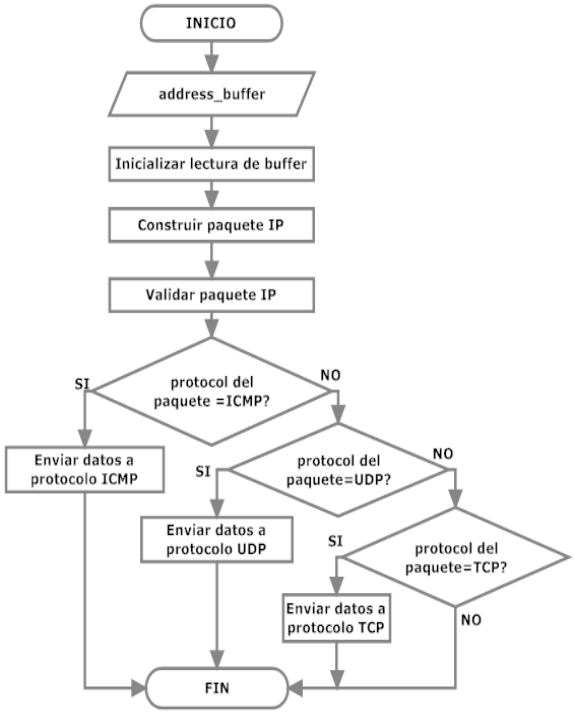


Figura 5.12 Algoritmo de recepción de datos IP

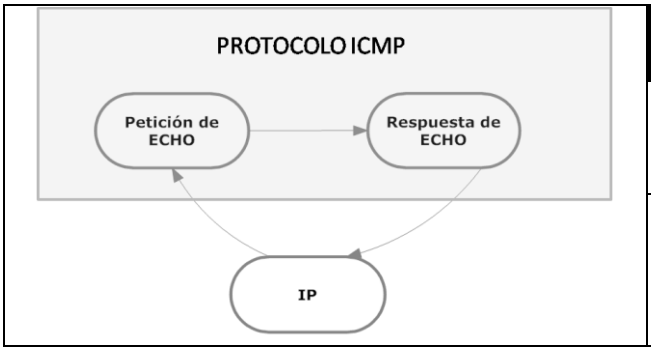
5.5 Protocolo de Mensajes de Control de Internet

El diseño del protocolo ICMP se basa en el RFC 792 [22e] y su implementación en esta pila consiste únicamente en la recepción de mensajes ECHO REQUEST y el envío de mensajes ECHO REPLY (envío de petición y envío de respuesta a la petición), de manera que el sistema cuente con la posibilidad de responder a comandos *ping* enviados desde cualquier sistema externo, con la finalidad de realizar pruebas de comunicación entre los equipos. En la Tabla 5.10 se presentan las acciones de recepción de solicitudes y envío de respuestas que realiza el protocolo ICMP y el nombre que se eligió para la función que implementa cada acción, así como un diagrama de la interacción de las funciones del protocolo ICMP con IP, ya que es el único protocolo con el que interactúa para enviar y recibir mensajes.

La estructura que se utiliza para almacenar los paquetes con formato ECHO ICMP se muestra a continuación y corresponde a la ilustración de la Figura 4.14.

```
struct icmp_echo_message{
  unsigned char type;     // 8 echo request; 0 echo reply
  unsigned char code;
  unsigned int  checksum;
  unsigned int  id;
  unsigned int  secnum;
  unsigned int  datalen;
};
```

Tabla 5.10 Funciones del protocolo ICMP



Acción que realiza	Nombre de la función
Recepción de peticiones	icmp_receive().
Envío de respuestas	icmp_send_echo_response()

5.5.1 Recepción de solicitudes ICMP ECHO REQUEST

La función de recepción ICMP se denomina **icmp_receive()**. Se llama desde el software IP cuando procesa un paquete cuyo campo *protocolo* corresponde al código del protocolo ICMP. Recibe como argumentos un número entero que indica la posición de los datos ICMP en el buffer de recepción, la longitud de los datos ICMP que contiene el buffer y la dirección IP del dispositivo remoto que envía la solicitud. No tiene valor de retorno. Para llamar a esta función desde IP:

icmp_receive(icmp_addrs_buff, len, ip);

Al igual que en los protocolos anteriores, primero se usa la función del controlador de red que inicializa el buffer de recepción para su lectura a partir de la posición de los datos del mensaje ICMP. Una vez hecho esto se utilizan las funciones de lectura del controlador para recuperar los datos correspondientes a los campos del mensaje y almacenarlos en una variable struct icmp_echo_message. Después se valida el contenido, incluyendo la suma de verificación, si la validación falla en cualquier punto, se sale de la función sin procesar los datos. Si la validación tiene éxito se examina el campo *tipo* para verificar que corresponda a un mensaje de solicitud, si es así se envía una respuesta.

5.5.2 Envío de mensajes ICMP ECHO REPLY

La función de envío de respuestas ECHO ICMP es llamada desde la función de recepción ICMP, se denomina **icmp_send_echo_response()** y el único argumento que recibe es la dirección IP del dispositivo remoto a quien le envía la respuesta. No tiene valor de retorno. Su implementación es bastante sencilla, se forma el mensaje de respuesta, se calcula el valor del checksum y se envía a través de la función ip_send().

5.6 Protocolo de Datagramas de Usuario (UDP)

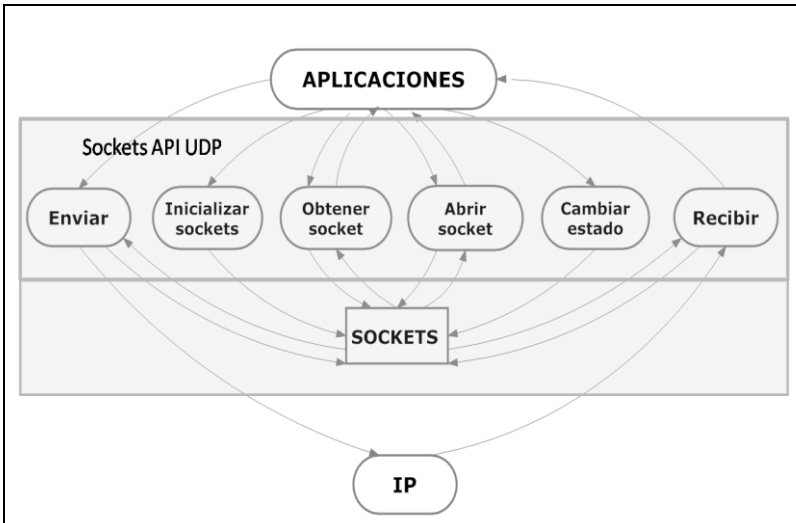
El diseño del protocolo UDP está basado en las recomendaciones del RFC 768 [23e] y su implementación conlleva acciones para el manejo de *sockets* y para el envío y recepción de datos. Las acciones a las que se hace referencia corresponden a las explicadas en el tema 4.8 y 4.10. En la Tabla 5.11 se presentan dichas acciones y el nombre que se eligió para la función que implementa cada acción, así como un diagrama de la interacción de las funciones del protocolo.

A continuación se muestran las estructuras correspondientes al datagrama de usuario ilustrado en la Figura 4.18 y la estructura de un socket UDP.

```
//datagarama udp
struct udp_datagram{
  unsigned int local_port;
  unsigned int remote_port;
  unsigned int message_len;
  unsigned int checksum;
  unsigned int data;
};
```

```
//socket udp
struct udp_sock{
   unsigned char state;
   unsigned char type;
   unsigned int  localport;
   unsigned int  remoteport;
   unsigned char remoteip[4];
   void(*udp_application)(char, unsigned int, unsigned int);
};
```

Tabla 5.11 Funciones del protocolo UDP y manejo de sockets



Acción que realiza	Nombre de la función
Inicialización de sockets	udp_socket_init()
Obtención de un socket	udp_get_sock()
Abrir un socket	udp_open_sock()
Cambiar estado del socket	udp_change_sock_state()
Envío de datos	udp_send()
Recepción de datos	udp_receive()

El socket UDP almacena el estado en que se encuentra el socket (libre, abierto o cerrado), el tipo de socket (socket cliente o socket servidor), los valores de los puertos local y remoto, la dirección IP remota y la referencia a una función de manejo de datos de entrada, definida por cada aplicación que haga uso de UDP.

5.6.1 Manejo de sockets UDP

Se define un conjunto de funciones empleadas por las aplicaciones que corren sobre UDP, mediante las cuales tienen la posibilidad de hacer uso de un socket para poderse comunicar a través de él. El software UDP declara un arreglo de datos tipo struct udp_sock que ofrece un conjunto finito de estos elementos a las aplicaciones. Cada aplicación puede hacer uso de un socket siempre y cuando existan sockets disponibles. Las funciones de manejo de sockets operan sobre dicho arreglo.

Inicialización de sockets. La función de inicialización recorre el arreglo de sockets ajustando su estado a libre (FREE). Se llama desde cualquier aplicación que use UDP de la siguiente manera:

udp_socket_init();

Puede ser invocada al inicio del programa principal como parte de la inicialización general del sistema, o dentro de alguna de las funciones de la aplicación, sólo se debe llamar una vez antes de realizar cualquier tipo de recepción o transmisión de datos. No recibe argumentos ni tiene valor de retorno.

Obtención de un socket. Cuando una aplicación requiere realizar comunicación a través de UDP debe invocar a la función **udp_get_sock()** para obtener un socket. Esta función recorre el arreglo de sockets examinando su estado, si encuentra uno libre regresa el índice del arreglo a la aplicación, con el cual ésta podrá manipular el socket a través de alguna de las funciones de manejo de sockets. En caso de recorrer todo el arreglo y no encontrar sockets en estado libre, regresa un entero negativo a la aplicación. La función se invoca de la siguiente manera, donde *socket* es una variable tipo entero, no recibe argumentos:

socket = udp_get_sock();

Abrir un socket. Una vez obtenido un socket válido para la comunicación, éste debe abrirse antes de realizar cualquier transmisión o recepción de datos, usando la función **udp_open_sock()**. Al abrir un socket se le deben enviar a la función los siguientes argumentos:

- Identificador de socket que se está abriendo
- Tipo de socket (cliente o servidor)
- Dirección IP del dispositivo remoto con el que se desea comunicar
- Puerto remoto
- Puerto local
- Nombre de la función que maneja los datos de entrada

La Figura 5.13 ilustra el algoritmo para abrir un socket UDP. La función valida los argumentos recibidos para saber si es posible abrir un socket con los datos solicitados. En caso de intentar abrir un socket servidor se debe especificar el puerto local por donde la aplicación espera recibir conexiones, así como una función de manejo de datos de entrada, si alguno de estos dos parámetros es cero, se sale de la función sin abrir el socket. Los argumentos de IP y puerto remoto pueden ser cero, ya que se definirán al momento que un cliente se conecte con la aplicación servidor UDP. Si el socket que se intenta abrir es de tipo cliente, udp_open_sock() valida que los parámetros IP y puerto remoto no sean cero, ya que indican a qué aplicación remota se van a enviar datos. El puerto local y la función de manejo de datos de entrada pueden ser cero si no se espera recibir respuesta de parte de la aplicación remota. Si la validación resulta correcta se ajustan los datos del socket con los valores recibidos y su estado se cambia a abierto (OPEN). No tiene valor de retorno.

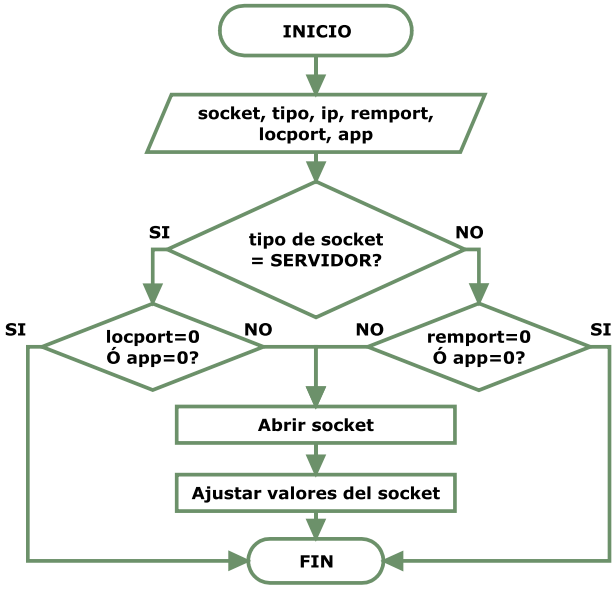


Figura 5.13 Apertura de socket cliente o servidor UDP

Para llamar a esta función desde una aplicación servidor, donde *app* es el nombre de una función definida en la aplicación que se encarga de procesar los datos recibidos desde UDP:

```
localport=2002;
udp_open_sock(socket, UDP_SERVER, 0, 0, localport, app);
```

El ejemplo anterior indica que tras obtener un socket UDP éste se abre como servidor, esperando conexiones a través del puerto 2002. El término "conexiones" no es correcto en UDP, ya que, como se vio antes, UDP es un protocolo no orientado a la conexión, lo que implica que no mantiene comunicación constante de manera implícita con el dispositivo remoto. Un socket se puede abrir para enviar y recibir datos de diferentes destinos mientras esté abierto. El socket UDP no está asociado a una sola aplicación remota. Sin embargo al decir conexión nos referimos al intercambio de información con un destino específico en un momento dado.

Para una aplicación que desea abrir un socket UDP como cliente, la función se invoca de la siguiente manera (después de haber obtenido un identificador válido de socket), los argumentos localport y app pueden ser cero.

```
ip[]={192,168,2,3};
rem_port = 2000;
localport = 2002;
udp_open_sock(socket, UDP_CLIENT, ip, rem_port, localport, app);
```

Cerrar un socket. Para cerrar un socket se debe invocar a la función udp_change_sock_state(socket,CLOSED), enviarle el identificador del socket que se desea cerrar y el nuevo estado que se requiere.

Liberar un socket. Cuando ya no es necesario el uso del socket se debe liberar para permitirle ser usado por otra aplicación. Esto se hace llamando a la función udp_change_sock_state(socket,FREE) a la cual se le envía el identificador del socket que se desea liberar y el nuevo estado que se requiere.

5.6.2 Procesamiento de salida de datos

La función de UDP utilizada por las aplicaciones para enviar datos se denomina **udp_send()**. Los parámetros que recibe son: el identificador de socket usado para la comunicación, la dirección de un buffer que contiene los datos a transferir y la longitud de los datos. Al llamar a esta función el socket debe estar abierto y contener una dirección IP y puerto remoto válido. Si el socket se abrió como cliente estos datos se especificaron al abrir el socket, si se abrió como servidor y se está usando la función de transmisión, es porque un cliente se conectó con el servidor y requiere una respuesta, por lo tanto en la función de recepción se almacenó la dirección IP y puerto remoto en el socket. Esta función se llama de la manera siguiente:

```
datos[]="Mensaje enviado a través de UDP";
udp_send(socket,datos,31);
```

udp_send() añade a los datos los encabezados del protocolo, tomando los datos almacenados en el socket y los envía a través de ip usando la función ip_send().

5.6.3 Procesamiento de entrada de datos

Cuando se recibe un paquete de datos desde IP la información se procesa en la función **udp_receive()**, que recibe como argumentos un número entero que indica la posición de los datos udp en el buffer de recepción, su longitud y la IP de quien los envía. Al igual que en los protocolos anteriores, la recepción comienza con la inicialización del buffer RX para leer los datos a partir de la posición donde se encuentran los datos de UDP, después de esto se leen los datos y se forma el paquete UDP con una variable tipo struct udp_datagram. Luego se valida la información recibida. El *checksum* no se implementa en esta pila (ni en la recepción ni en la transmisión) para mejorar el desempeño y rapidez de la comunicación y debido a que la suma de verificación no es un mecanismo muy eficiente de detección de errores. Si la validación se realiza satisfactoriamente se procede a buscar un socket asociado a una aplicación local para entregar los datos, comparando el número de puerto local del socket con el número de puerto remoto del paquete recibido. Si se encuentra alguna coincidencia la información se envía a la función especificada en el socket y se salva la información del dispositivo remoto en el socket. Para llamar a la función de recepción UDP desde el software IP:

```
udp_receive(udp_addrs_buff, datalen, ip);
```

EJEMPLO

A continuación se muestra el procedimiento que debe seguir una aplicación cliente que desea comunicarse con una aplicación remota a través del protocolo UDP.

```
unsigned char ip[]={192,168,2,3};
unsigned int rem_port = 2000;
unsigned int localport = 2002;
unsigned char datos[]= "Mensaje enviado a través de UDP";
```

```
//inicialización del pool de sockets
udp_socket_init();

//obtención de un socket para la comunicación
socket = udp_get_sock();
if(socket<0)
{
  error;
  return;
}

//abrir el socket tipo cliente
udp_open_sock(socket, UDP_CLIENT, ip, rem_port, localport, app);

//envío de los datos
udp_send(socket,datos,31);
```

Para una aplicación servidor se abre el socket como se mostró antes y se espera hasta la recepción de datos. En el capítulo 7 se muestra la programación de una aplicación para el microcontrolador haciendo uso de las funciones descritas en este tema, ahí se muestra la función de manejo de datos de entrada de la aplicación.

5.7 Protocolo de Control de Transmisión (TCP)

El diseño del protocolo TCP se basa en las recomendaciones del RFC 793 [24e]. El software del protocolo se divide en acciones para el manejo de sockets, envío y recepción de datos y una rutina periódica llamada desde la función principal. En Tabla 5.12 se presentan dichas acciones y el nombre que se eligió para la función que implementa cada acción. Las acciones están basadas en la descripción del protocolo TCP del tema 4.9 y la descripción del uso de sockets del tema 4.10.

Tabla 5.12 Funciones del protocolo TCP y manejo de sockets

Acción que realiza	Nombre de la función
Inicialización de sockets	tcp_socket_init()
Obtención de un socket	tcp_get_sock()
Obtener estado del socket	tcp_get_sockstate()
Abrir conexión pasiva	tcp_sock_listen()
Abrir conexión activa	tcp_sock_connect()
Cerrar conexión	tcp_close()
Reiniciar conexión	tcp_reset()
Abortar una conexión	tcp_abort()
Liberar un socket	tcp_free_sock()
Envío de datos y confirmación	tcp_send() tcp_data_send()
Recepción de datos	tcp_receive()
Rutina periódica	tcp_poll()

En la Figura 5.14 se presenta un diagrama que muestra la interacción de las funciones del protocolo TCP con el conjunto de sockets, con otras funciones del protocolo y con los otros niveles. Más adelante se muestra la descripción de cada una de las funciones y se recomienda consultar el diagrama para facilitar la comprensión.

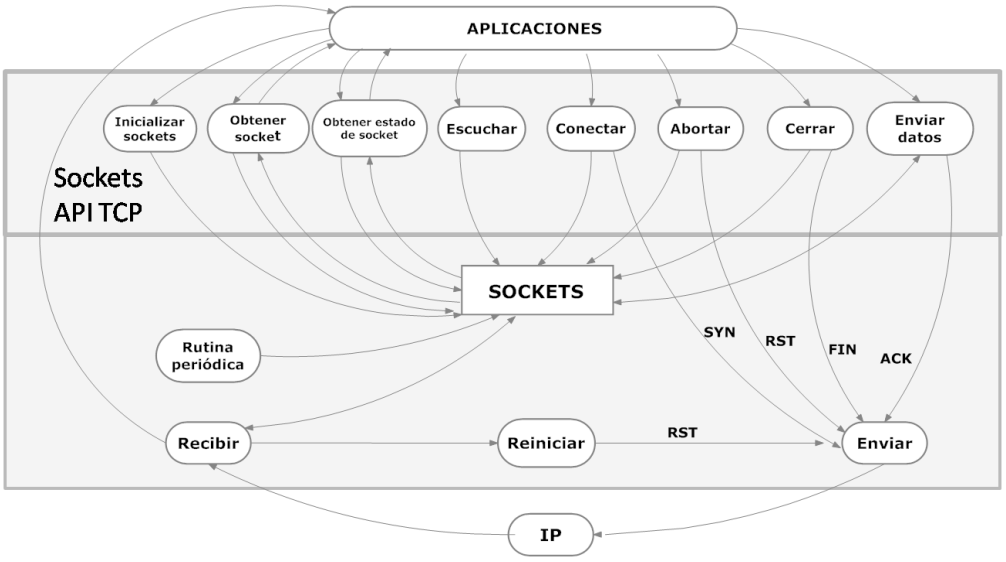


Figura 5.14 Funciones del protocolo TCP

A continuación se muestra la estructura empleada por TCP de acuerdo al formato de la Figura 4.25, así como la estructura que define un socket TCP, la cual se basa en las recomendaciones establecidas en el estándar RFC 793.

```
//paquete tcp
struct tcp_packet{
  unsigned int  localport;
  unsigned int  remport;
  unsigned long secnum;
  unsigned long acknum;
  unsigned int  hlen_flags;
  unsigned int  window;
  unsigned int  checksum;
  unsigned int  urgpoint;
  unsigned char tcpopt[TCP_LEN_BYTE(TCP_MAX_OPT)];
  unsigned int  data;
};

//socket tcp
struct tcp_sock{
  unsigned char state;
  unsigned char type;
  unsigned char remip[4];
  unsigned int  remport;
  unsigned int  localport;
  unsigned long secnum;
  unsigned long remsecnum;
  unsigned char data_unack;
  unsigned char retries;
  unsigned char closepending;
  unsigned char retransmitclock;
  unsigned char timetoliveclock;
  unsigned int  dlen;

  char (*application)(char, unsigned char);
};
```

El socket empleado por el protocolo TCP almacena muchas variables para guardar el estado de una conexión, los datos del dispositivo remoto y los relojes necesarios. El *estado* indica en qué etapa de la

comunicación se encuentra la conexión. El tipo especifica si se trata de un socket cliente o servidor. *remip* y *remport* indican la dirección de red y de puerto, respectivamente, del dispositivo remoto. *localport* es el identificador del puerto local. La variable *secnum* almacena el número de secuencia local empleado por el socket, se incrementa después de cada transmisión. *remsecnum* guarda el número de secuencia remoto esperado por parte del dispositivo con el que se está comunicando, se incrementa cada vez que se reciben segmentos TCP de parte del dispositivo remoto y se envía una confirmación. La variable *data_unack* sólo guarda un 1 o un 0 que indican si se ha recibido o no la confirmación de datos previamente enviados. *data_unack* se examina cada vez que se desea enviar datos, en caso de que el socket tenga datos no confirmados, no se realiza el envío. *retries* es una variable que guarda el número de retransmisiones permitidas a un socket cuando no recibe la confirmación de datos enviados, se decrementa cada vez que se realiza una retransmisión, al alcanzar el valor de cero se cierra la conexión por no recibir respuesta del equipo remoto. *closepending* es una bandera que almacena el valor 0 o 1 e indica que hay una petición de término de conexión no atendida y que el socket se debe cerrar en cuanto sea posible. Las dos variables siguientes son enteros que se relacionan con temporizadores para hacer operaciones de tiempo, *retransmitclock* se inicia cada vez que se envían datos y se usa para realizar retransmisiones en caso de que llegue a 0 y la confirmación no se haya recibido. *timetoliveclock* se inicia cada vez que se envía o recibe algo en el estado de CONECTADO, si pasa demasiado tiempo sin que haya intercambio de información entre ambas partes de la comunicación y este reloj vence, la comunicación se cierra por haber estado inactiva por mucho tiempo. Por último la variable *dlen* almacena la longitud de los datos de aplicación transportados por el socket, ya sea en el envío o en la recepción (al decir datos de aplicación significa que no se consideran los datos de los encabezados TCP, un segmento de control contiene una longitud de datos igual a cero).

5.7.1 Máquina de estados TCP

El algoritmo TCP se puede explicar como una máquina de estados que indica el estado en el que se encuentra la comunicación y las transiciones se definen por las banderas enviadas y recibidas o por la invocación de funciones del protocolo, de acuerdo al algoritmo explicado en el tema 4.9 (ver ejemplo 4.9.2.4). Los estados de la comunicación son los que se definen a continuación y corresponden a las recomendaciones del RFC 793 [24e] para el protocolo TCP:

FREE. Los sockets no están en uso y se encuentran disponibles para ser usados por cualquier aplicación.

CLOSED. El socket se encuentra en estado cerrado cuando se obtiene y aun no se ha realizado ninguna acción o cuando se finaliza la comunicación.

LISTENING. Cuando se abre una conexión pasiva en un socket servidor, su estado corresponde a LISTENING.

SYN_SENT. Cuando una aplicación cliente desea establecer comunicación con un servidor abre una conexión activa enviando un segmento TCP con la bandera de SYN y su número de secuencia inicial. Después de haber enviado dicho segmento, cambia su estado a SYN SENT (SYN enviado), que le indica a la máquina de estados TCP que el próximo paquete que espera recibir debe contener las banderas SYN y ACK y el número de secuencia inicial del dispositivo remoto.

SYN_RECEIVED. Cuando un servidor se encuentra en estado LISTENING y recibe una solicitud de conexión, envía un segmento TCP con las banderas SYN y ACK y cambia su estado a SYN_RECEIVED (SYN recibido), indicando que el próximo segmento que espera recibir debe tener la confirmación de su respuesta para establecer la conexión.

CONNECTED. Tanto el cliente como el servidor se encuentran en estado CONNECTED una vez que han completado las tres etapas del establecimiento de la conexión. En este estado pueden intercambiar información en ambas direcciones.

FIN_WAIT1. Cuando uno de los dos extremos de la conexión decide terminar la comunicación mientras se encuentra en estado CONNECTED, envía un segmento con la bandera de FIN y cambia su estado a FIN_WAIT1, indicando que espera recibir la confirmación de su petición.

FIN_WAIT2. Cuando la comunicación está en el estado FIN_WAIT1 y se recibe la confirmación de la desconexión, ya no se pueden enviar más datos, sólo confirmaciones mientras el otro extremo continúe

abierto. Este estado se indica mediante FIN_WAIT2 e indica que aún se espera la solicitud de desconexión remota para finalizar la comunicación.

CLOSE_WAIT. Indica que se espera recibir una solicitud de finalización proveniente de la aplicación local, ya que la comunicación por parte del dispositivo remoto se ha cerrado.

CLOSING. Cuando se ha enviado una solicitud de desconexión de la cual no se ha recibido la confirmación pero ya se recibió la solicitud de desconexión del dispositivo remoto, se cambia el estado a CLOSING indicando que se espera recibir una confirmación.

LAST_ACK. Cuando ambas partes han enviado su solicitud de desconexión y sólo una de ellas ha sido confirmada, el estado de la comunicación corresponde a LAST_ACK, indicando que sólo se espera recibir la última confirmación de desconexión para cerrar definitivamente la conexión.

TIME_WAIT. Una vez cerrada la conexión, el estado corresponde a TIME_WAIT, lo que indica que se debe dejar transcurrir algún tiempo antes de pasar al estado CLOSED para asegurar que la última confirmación llegó a su destino y no es necesario enviarla de nuevo.

Para que la idea de la máquina de estados TCP quede clara se muestra de manera esquemática en la Figura 5.15, donde se aprecia cada estado y las acciones que generan transiciones entre uno y otro. El diagrama también se tomó del RFC 793 [24e].

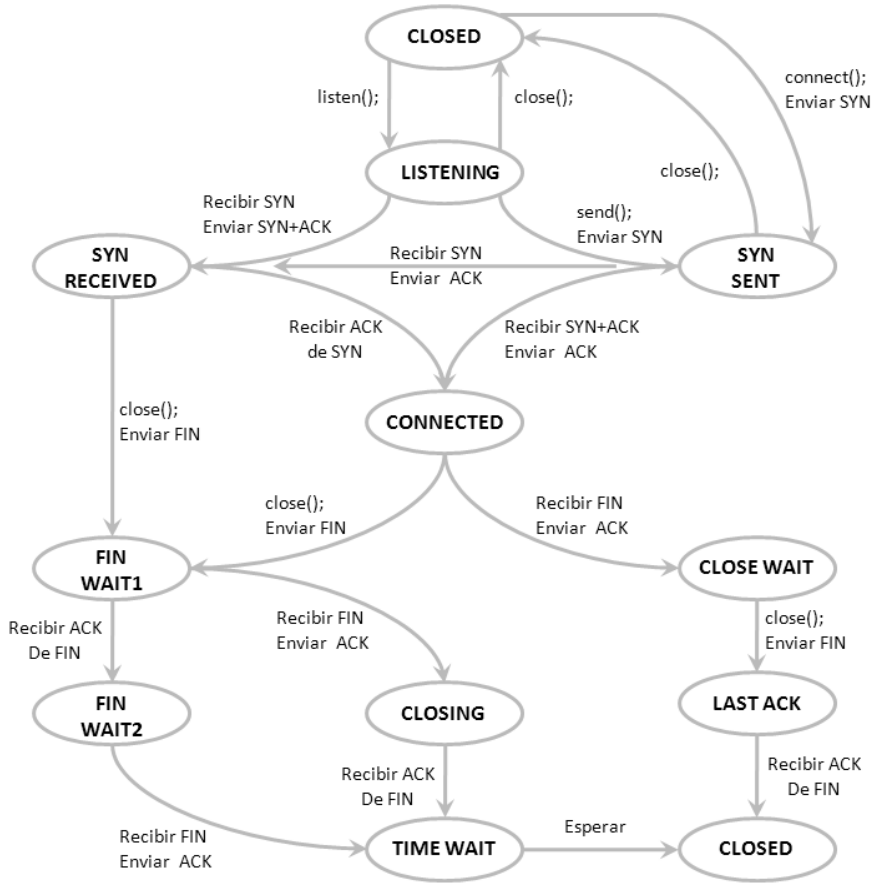


Figura 5.15 Máquina de estados TCP

5.7.2 Manejo de sockets y conexiones TCP

Las aplicaciones que emplean TCP disponen de una interfaz de funciones que operan sobre un conjunto de sockets definidos en el software TCP mediante un arreglo de datos tipo struct tcp_sock. A través de estas

funciones las aplicaciones pueden obtener un socket y configurarlo como cliente o servidor para realizar envío y recepción de datos con aplicaciones remotas. Un ejemplo de aplicación que emplea sockets TCP es HTTP, explicada en el tema 5.8.

Inicialización de sockets. La función de inicialización recorre el arreglo de sockets ajustando su estado a libre (FREE). Se llama desde cualquier aplicación que use TCP de la siguiente manera:

```
tcp_socket_init();
```

Puede ser invocada al inicio del programa principal como parte de la inicialización general del sistema, o dentro de alguna de las funciones de la aplicación, sólo se debe llamar una vez antes de realizar cualquier tipo de recepción o transmisión de datos. No recibe argumentos ni tiene valor de retorno.

Obtención de un socket. Cuando una aplicación requiere realizar comunicación a través de TCP debe invocar a la función **tcp_get_sock()** para obtener un socket. Los parámetros que recibe son el tipo de socket y una referencia a una funcion de procesamiento de datos de entrada (definida en la aplicación). Primero valida los argumentos recibidos y después recorre el arreglo de sockets examinando su estado. Si encuentra uno libre regresa el índice a la aplicación, con el cual ésta podrá manipularlo a través de alguna de las funciones de manejo de sockets. En caso de recorrer todo el arreglo y no encontrar sockets en estado libre, regresa un entero negativo. La función se invoca de la siguiente manera, donde *socket* es una variable tipo entero y *app* es el nombre de una función de la aplicación que procesa los datos recibidos:

```
socket = tcp_get_sock(SERVER,app);
```

En el tema 5.8 se muestra un ejemplo del empleo de esta función y la función de la aplicación que procesa los datos recibidos.

Obtener el estado de un socket. La función tcp_get_sockstate() se utiliza para obtener el estado en el que se encuentra un socket. Recibe como parámetro el identificador de socket evaluado y regresa su estado. Para llamar a esta función desde una aplicación que haga uso de TCP:

```
state = tcp_get_sockstate(socket);
```

Abrir conexión pasiva. Cuando una aplicación TCP se configura como servidor, al abrir el socket se dice que está "escuchando", o que realiza una conexión pasiva. La función que realiza esta acción se denomina tcp_sock_listen(). Los parámetros que recibe son el identificador de socket que se requiere abrir y el número de puerto local por el que espera recibir conexiones. No tiene valor de retorno. La función verifica que tanto el identificador de socket como el número de puerto sean válidos, que el socket sea tipo cliente y que esté cerrado. Si todo lo anterior es correcto, cambia el estado del socket a LISTENING y guarda el número de puerto local. Esta función se invoca de la siguiente manera:

```
tcp_sock_listen(socket, localport);
```

Abrir conexión activa. Cuando una aplicación TCP se configura como cliente se usa la función tcp_sock_connect() para establecer comunicación con un servidor. Los parámetros que recibe son el identificador de socket, la IP y puerto remotos y el número de puerto local que se va usar durante la comunicación. La función valida que los parámetros recibidos sean diferentes de cero, que el socket sea de tipo cliente y que se encuentre cerrado. Si los datos se validan de forma correcta se guardan en el socket los datos del dispositivo remoto y el número de puerto local, después se envía un segmento TCP con bandera de SYN y se cambia el estado a *syn enviado* (SYN_SENT). Finalmente se incrementa el número inicial de secuencia que corresponde al número del primer segmento de datos que se enviará y se inicia el reloj de retransmisión. La función se invoca de la siguiente manera:

```
ip[]={192,168,2,3};
rem_port = 80;
localport = 2002;
tcp_sock_connect(socket, ip, rem_port, localport);
```

Fin de la conexión. Cuando se decide que es tiempo de terminar la comunicación, se envía una solicitud de desconexión usando la función **tcp_close(socket)**. Esta función recibe un indicador de socket y regresa un entero que indica en qué estado se encuentra la comunicación después del intento de desconexión, ya que

podemos encontrarnos con varios casos. Si el socket que se intenta cerrar se encuentra ya cerrado o se acaba de enviar una solicitud de conexión, se cierra cambiando su estado a CLOSED. Si el socket previamente recibió una solicitud de conexión y envió una confirmación, por lo que se encuentra en estado SYN_RECEIVED, se envía un segmento con solicitud de desconexión (bandera FIN), se ajustan las variables del socket (relojes, número de retransmisiones, número de confirmación) y se cambia al estado FIN_WAIT1. Si la comunicación se encuentra en cualquiera de los estados FIN_WAIT1, FIN_WAIT2, TIME_WAIT, LAST_ACK o CLOSING, no se realiza ninguna acción, ya que la desconexión ya se encuentra en proceso. Si el estado del socket es CONNECTED se verifica si hay datos pendientes por confirmar, en caso negativo se envía el segmento con solicitud de desconexión y se cambia el estado a FIN_WAIT1. En caso de que haya datos pendientes por confirmar se indica con la bandera de *cierre pendiente* y se espera a que otra rutina determine el momento de cerrar la conexión. Esta función debe ser invocada por la aplicación. La Figura 5.16 ilustra el algoritmo.

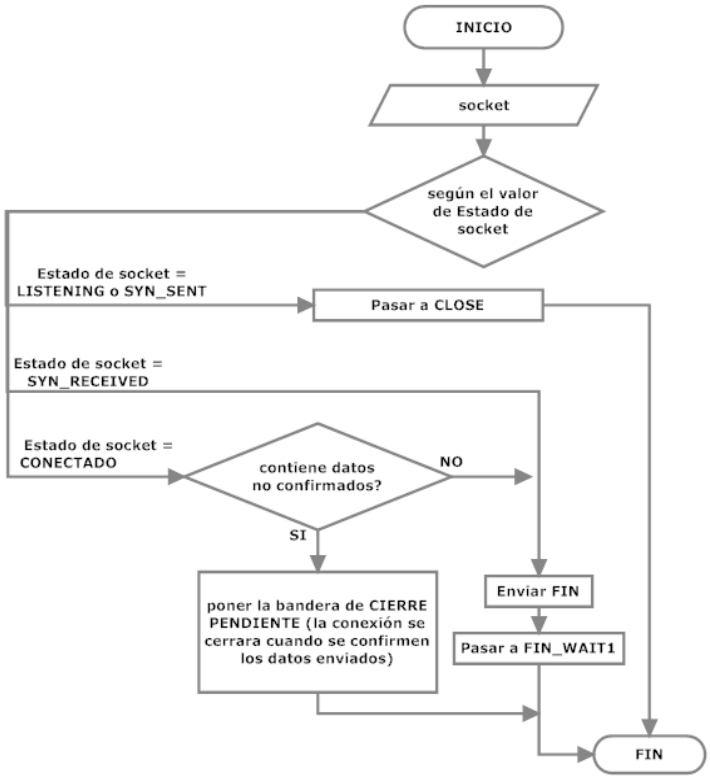


Figura 5.16 Función de fin de conexión TCP

Abortar y reiniciar una conexión. Cuando se suscita algún evento que implique el término inmediato de la comunicación (como alguna situación de error), se envía un segmento TCP con la bandera de reset (RST), que le indica al dispositivo remoto que la comunicación será interrumpida abruptamente; tras enviar este segmento el emisor pasa a su estado CLOSED y tras recibirlo el emisor hace lo mismo. Este caso es diferente a un cierre de conexión ya que no se respeta el procedimiento usual dictado por el protocolo, en el cual se tienen que intercambiar varios segmentos y la desconexión se da como un acto acordado por ambas partes en momentos independientes. Un ejemplo de esta situación se observa en el diagrama de la Figura 5.17 que ilustra el algoritmo de la función periódica, cuando un segmento enviado no recibe una respuesta y agota sus retransmisiones, cierra la comunicación enviando un paquete de reset.

Existen dos funciones que permiten abortar una conexión. La primera permite que se envíe un segmento con bandera RST desde otras funciones del protocolo, donde alguna situación no permite el procesamiento de la información (por ejemplo la recepción de un paquete con información desconocida). La segunda función es para uso de las aplicaciones en caso de que ocurra alguna falla inesperada a nivel de aplicación.

La función **tcp_reset()** es la que pueden usar otras funciones del protocolo. Recibe como argumento una variable tipo struct tcp_packet y la dirección IP del dispositivo remoto. Hay que notar la diferencia con otras funciones del protocolo al no recibir un identificador de socket, ya que al usarse bajo condiciones

anormales, es posible que se necesite enviar un reset a un dispositivo que envía información incorrecta, la cual no tiene ningún socket asociado, por lo tanto dicho socket no existe y no se puede utilizar como argumento. La función toma los datos recibidos y ensambla un paquete TCP, utiliza la función tcp_send() para enviar un segmento con bandera RST al dispositivo que envía información y pasa al estado CLOSED. No tiene valor de retorno. Se invoca desde otras funciones del protocolo TCP como se muestra a continuación, la variable tcp_pack contiene los datos del dispositivo que envió la información:

```
struct tcp_packet *tcp_pack;  // paquete tcp ensamblado en alguna otra parte
ip[]={192,168,2,3};           //ejemplo de IP remota
tcp_reset(tcp_pack, ip);
```

La función que permite interrumpir la comunicación desde la aplicación se denomina **tcp_abort()**. Esta función recibe un identificador de socket, a diferencia de la función anterior, aquí se considera que existe un socket asociado a la comunicación y que a través de éste ha estado fluyendo información de manera regular, pero debido a una situación anormal que la aplicación no puede manejar, se aborta la comunicación. La función valida el estado del socket y utiliza la función tcp_send() para enviar un segmento con bandera RST. Después cambia el estado del socket a CLOSED. No regresa ningún valor. La aplicación la debe invocar de la siguiente manera:

```
tcp_abort(socket);
```

Liberar un socket. Cuando ya no es necesario el uso del socket se debe liberar para permitirle ser usado por otra aplicación. Esto se hace llamando a la función **tcp_free_sock(socket)** a la cual se le envía el identificador del socket que se desea liberar y la función cambia su estado a libre (FREE).

5.7.3 Procesamiento de salida de datos

Se definen dos funciones de envío TCP: **tcp_send()** y **tcp_data_send()**. tcp_send() es la función encargada de ensamblar un paquete TCP tomando la información contenida en el socket y enviarla a la capa inferior a través de la función *ip_send()*. Dicha información se debe configurar previamente en la función que invoca a tcp_send(). Se invoca dentro del software TCP por otras funciones, por ejemplo para enviar segmentos de control con banderas como SYN, ACK o FIN. Los parámetros que recibe son el identificador de socket, la bandera y la dirección del buffer que contiene los datos. La información que se toma del socket para ensamblar el paquete y que debe ser ajustada antes de llamar a la función es el puerto local y puerto remoto, la dirección IP remota, los números de secuencia local y remoto y la longitud de los datos. Después de ensamblar el paquete TCP, agregando los encabezados a los datos recibidos, se calcula la suma de verificación, se añade en el campo adecuado y finalmente se llama a la función ip_send() para su transmisión. Regresa un entero que indica si el procesamiento fue adecuado o por algún error no se pudo completar. Para llamar a esta función desde cualquier otra del protocolo:

```
datos[]="Mensaje enviado a través de TCP";
tcp_socket[socket].dlen =  31;
tcp_send(socket, ACK, datos);  //segmento con datos y confirmación

datos[];
tcp_socket[socket].dlen =  0;
tcp_send(socket, SYN, datos);  //segmento de control con solicitud de conexión
```

La función tcp_data_send() es la que utilizan las aplicaciones para enviar datos a través de una conexión TCP abierta. Recibe el identificador de socket, la dirección del buffer que almacena los datos y su longitud. Antes de enviar los datos la función valida que el socket se encuentre en el estado CONNECTED y que no contenga datos no confirmados. Una vez realizadas las validaciones de manera exitosa, se hace uso de la función tcp_send() para enviar el paquete y se ajustan los valores del socket: el número de secuencia remoto se incrementa, se modifica la bandera de *datos no confirmados*, se reinician las retransmisiones a su valor máximo y se reinicia el reloj de retransmisión y el de conexión activa. Devuelve un entero que indica si la transmisión se realizó de manera adecuada o si ocurrió algún error y no fue posible completarse. Las aplicaciones deben invocar a esta función de la siguiente manera:

```
datos[]="Mensaje enviado a través de TCP";
tcp_data_send(socket,datos,31);
```

5.7.4 Procesamiento de entrada de datos

La función de recepción TCP se denomina **tcp_receive()**, se llama desde la rutina de recepción del protocolo inferior (IP) cuando éste decide que el paquete que procesó está destinado a TCP. Los argumentos que recibe son: la ubicación de los datos TCP en el buffer de recepción, su longitud y la dirección IP de quien envía la información. Esta función ensambla una variable tipo struct tcp_packet donde almacena los valores de los encabezados del paquete TCP, leyéndolos del buffer de recepción. Después realiza una validación del contenido de los encabezados, incluyendo la suma de verificación. A continuación busca un socket asociado a la comunicación, buscando cuál de ellos guarda la IP del dispositivo remoto, o cuál escucha a través del puerto indicado en el paquete recibido (en caso de ser un intento de establecer comunicación). Si no hay ningún socket asociado a la aplicación remota se envía un segmento de reset mediante la función tcp_reset(), considerando que ocurrió algún error. Si la validación resulta correcta y se encuentra un socket se implementa el algoritmo de la máquina de estados ilustrada en la Figura 5.15 mediante una estructura *switch* que evalúa el estado del socket y realiza la acción indicada para cada caso, como puede ser enviar un segmento de control (mediante la función *tcp_send()*), o hacer una llamada a la función de recepción de datos indicada en el socket si se reciben datos para ser procesados por la aplicación. En cada uno de los casos se ajustan las variables del socket de manera adecuada. Si se encuentra información no consistente, se envía un segmento de reset por medio de la función tcp_reset().

5.7.5 Rutina periódica TCP

Como TCP es un protocolo que guarda el estado de una conexión y tiene a su cargo varias tareas necesarias para asegurar una entrega de datos confiable, requiere de una rutina que debe ser llamada de manera periódica para determinar si llega el momento de tomar acciones sobre conexiones en las que haya ocurrido algún problema inusitado, como enviar una retransmisión si no se ha recibido respuesta, cerrar una conexión que ha estado inactiva por mucho tiempo o cerrar una conexión que contiene una solicitud de desconexión no atendida. La función encargada de realizar esta tarea se denomina **tcp_poll()**, debe ser invocada desde la función principal cada cierto intervalo de tiempo. No recibe parámetros ni regresa valor alguno. El algoritmo se muestra en la Figura 5.17, se implementa con una estructura *switch*.

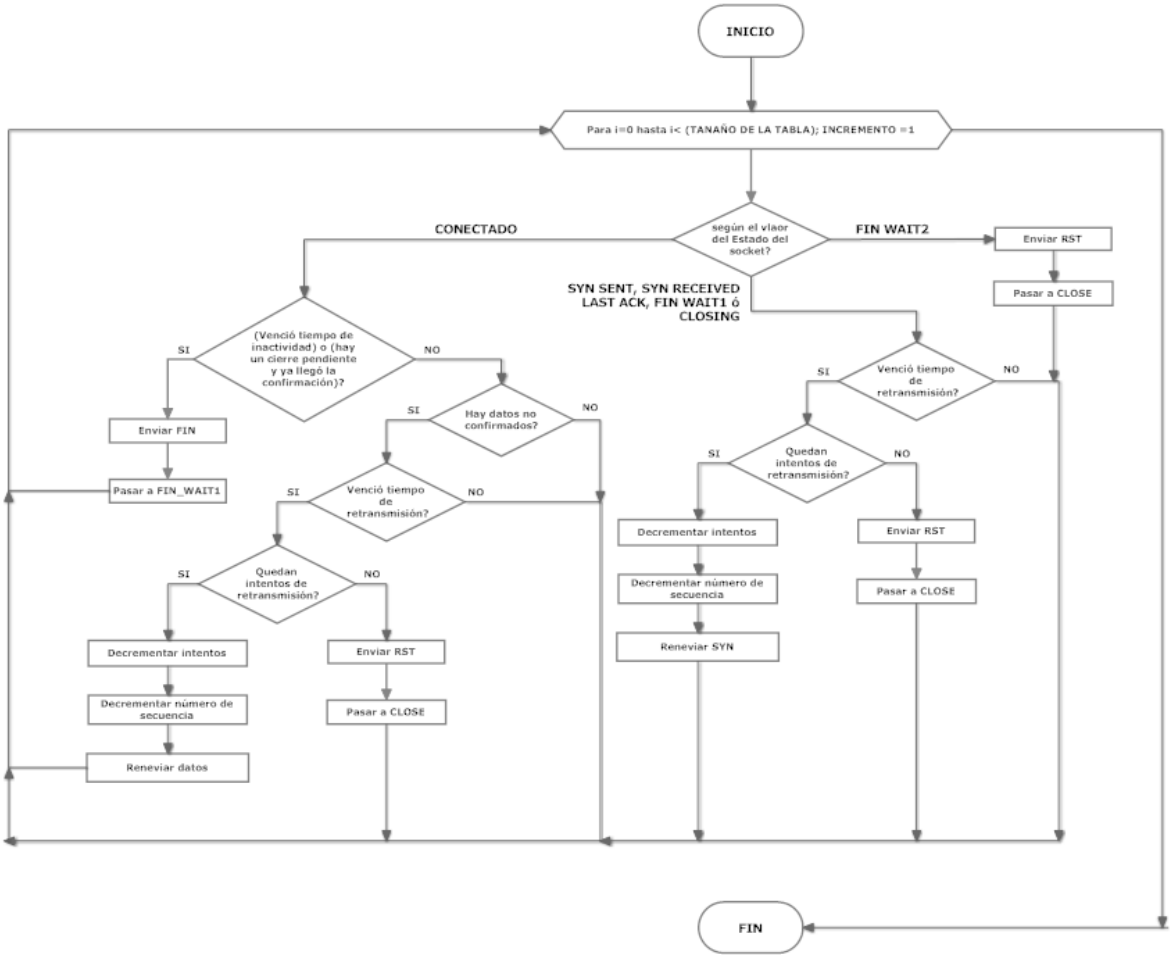


Figura 5.17 Rutina periódica del protocolo TCP

5.8 Protocolo de Transferencia de Hipertexto (HTTP)

El Protocolo de Transferencia de Hipertexto corresponde a un protocolo de la capa de aplicación programado sobre las funciones del protocolo TCP y su diseño está basado en las recomendaciones del RFC 2616 [25e]. En este proyecto se implementa la parte del protocolo correspondiente a la aplicación *servidor*, la aplicación *HTTP cliente* se puede ejecutar con cualquier navegador web desde cualquier computadora, por lo que no es necesario programarla para los fines que este proyecto pretende alcanzar. Si se deseara comunicar dos microcontroladores con relación cliente-servidor, entonces sería necesario implementar ambas partes del protocolo. Por el momento es suficiente la implementación de un servidor que corra en el microcontrolador, a fin de poder ser accedido desde cualquier navegador web habilitado en una computadora.

El servicio que se ofrece corresponde a la manipulación, monitoreo y control de un sistema a través de una página web que puede ser accedida por un determinado número de usuarios. En este tema sólo se aborda la transferencia de los recursos web a través de HTTP, la aplicación de control se explica a detalle en el capítulo 7.

Las acciones que realiza el servidor HTTP corresponden a las descritas en el tema 4.11. Para la implementación podemos dividir a las acciones en manejo de sesiones, envío, recepción y rutina periódica. Todas estas acciones están programadas sobre la interfaz que ofrece TCP. En la Tabla 5.13 se presentan las acciones que realizan el protocolo y el nombre que se eligió para la función que implementa a cada una.

Tabla 5.13 Funciones del protocolo HTTP

Acción que realiza	Nombre de la función
Inicialización	http_init_server()
Rutina periódica	http_run()
Procesar entrada de datos	http_listener()
Envío de datos	http_send(session)
Abrir una sesión	http_open_session()
Buscar una sesión	http_find_session()
Cerrar una sesión	http_close_session()

En el protocolo HTTP se define el concepto de *sesión* para identificar las diferentes conexiones realizadas a través de un mismo puerto TCP que brinda un servicio a varios usuarios simultáneamente. Mediante las sesiones HTTP administra la información enviada y recibida. La siguiente estructura se define en el software HTTP para implementar el concepto de sesión:

```
struct session_state{

  unsigned char state;
  unsigned int  socket;
  unsigned char *filestart;
  unsigned long filelen;
  unsigned long pointer;
  unsigned long datasent;
};
```

El estado de una sesión puede ser abierta o cerrada. Cuando la sesión no ha sido usada permanece cerrada pero cuando un cliente se conecta al servidor se utiliza una de las sesiones disponibles (a cada sesión se le asigna un socket TCP mediante la variable *socket*), cambia su estado a abierta y en ese momento nadie más puede usar esta sesión hasta que el cliente la deseche. El puntero *filestart* guarda la dirección del archivo a transferir. Por cada sesión que se abra se transfiere un archivo que se encuentra almacenado en una variable en la memoria flash del microcontrolador y cuando el archivo se termina de transferir la sesión se cierra. La longitud del archivo se almacena en la variable *filelen*. *Pointer* indica el último byte enviado, se incrementa cuando se recibe la confirmación y por tanto indica también la cantidad de bytes del archivo que se han transferido y confirmado (es muy probable que un recurso se transfiera en varios segmentos TCP). La última variable, *datasent*, guarda la cantidad de datos enviados en cada paquete TCP.

Supongamos que un archivo está compuesto de 100 bytes y que TCP tiene capacidad para enviar 10 bytes en cada paquete. La variable *filelen* contendría el valor de 100. Al enviar el primer segmento con 10 datos *datasent* valdría 10 y *pointer* cero. En el momento que se recibe la confirmación de los primeros 10 bytes, *pointer* se incrementa al valor de 10 y *datasent* vuelve a ser cero, indicando que por el momento no hay datos enviados. Al enviarse el siguiente segmento de datos, se forma un paquete TCP con los datos del archivo del 11 al 20, y *datasent* valdría 10 nuevamente. Si en este punto se pretendiera transferir otra porción del archivo no sería posible, ya que la variable *datasent* indica que hay datos enviados que aún no han sido confirmados. Nuevamente al llegar la confirmación pointer se incrementa al valor de 20, lo cual indica que los primeros 20 bytes ya fueron transferidos y se forma un nuevo paquete con los datos del 21 al 30. El procedimiento se repite hasta completar la transferencia del archivo completo, lo cual se identifica cuando la variable *pointer* alcanza el valor de la variable *filelen*.

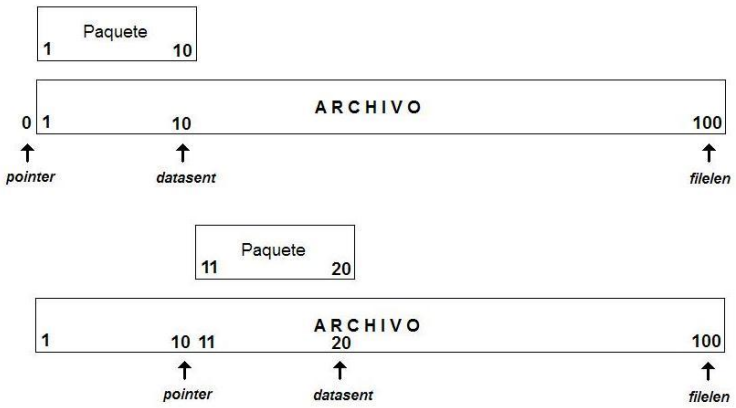


Figura 5.18 Transferencia de archivos HTTP

Para manejar los archivos almacenados en el servidor se emplea una estructura del siguiente tipo:

```
struct files{
  unsigned char       hash;
  const unsigned char *source_file;
  unsigned int        file_len;
};
```

La variable *hash* es un número entero que identifica a un archivo, se obtiene aplicando un algoritmo numérico a los caracteres del nombre del archivo. El apuntador *source_file* contiene la dirección del inicio del archivo y corresponde al nombre del arreglo que almacena todos los bytes del archivo. *file_len* contiene la longitud.

Para almacenar recursos web en el servidor se declara un arreglo de datos tipo struct files, por lo que cada elemento del arreglo contiene los datos de un archivo. Los archivos se codifican en ASCII y se almacenan en una variable. Cada vez que se añade un nuevo recurso al servidor, como puede ser una página, una imagen, o cualquier otro tipo de recurso, se le da un nombre, se aplica el algoritmo para obtener el *hash* a partir del nombre y se emplea un programa que codifica el documento en ASCII, genera la variable y la longitud del archivo. Estos datos se almacenan en el arreglo de recursos. En la Figura 5.19 se ilustra un ejemplo de un arreglo que contiene tres archivos: una página de error que indica que el recurso solicitado no se encuentra en el servidor, una página de inicio y una imagen. Debajo del arreglo se muestra un ejemplo de codificación de un recurso, la página *not_found* se muestra directamente programada en una variable y la imagen se encuentra codificada en ASCII. La codificación se obtiene ejecutando un programa que genera la variable mostrada (se muestra sólo una parte de la codificación generada).

```
const struct files resources[3];
```

	hash	*source_file	file_len
resources[0]	215	not_found_file	88
resources[1]	11	index_file	114
resources[2]	228	imagen_file	807

```
const unsigned char not_found_file [] = "<HTML><HEAD><TITLE> not found
</TITLE></HEAD><BODY> RECURSO NO ENCONTRADO </BODY></HTML>";

const unsigned char imagen_file [] = {
0xFF,0xD8,0xFF,0xE0,0x00,0x10,0x4A,0x46,0x49,0x46,0x00,0x01,0x01,0x01,
0x00,0x60,0x00,0x60,0x00,0x00,0xFF,0xDB,0x00,0x43,0x00,0x08,0x06,0x06,
0x07,0x06,0x05,0x08, ……
```

Figura 5.19 Ejemplo de archivos almacenados en el servidor

5.8.1 Manejo de sesiones HTTP

En el software HTTP se declara un arreglo de datos tipo *struct session_state*, que permite la creación de varias sesiones y su uso se efectúa a través de un conjunto de funciones.

Inicialización. La función de inicialización se denomina **http_init_server()**, no recibe argumentos ni tiene valor de retorno. Debe ser invocada desde la función principal, es la que habilita el servidor web que funciona en el microcontrolador. Esta función recorre el arreglo que guarda las sesiones, utiliza la función tcp_get_sock() para obtener un socket TCP libre para cada una de las sesiones. Después invoca a la función tcp_sock_listen() pasándole el identificador de socket obtenido antes y el valor numérico 80, ya que corresponde al puerto por el que escucha todo servidor web. El siguiente fragmento de código muestra cómo se inicializa el servicio en el servidor web.

```
for(i=0;i<HTTP_SESSIONS_NUM;i++){
      http_session[i].state = HTTP_CLOSED;
      sock=tcp_get_sock(TCP_SERVER, http_listener);
      if(sock<0)
        reset_code(); //ya no hay sockets disponibles
      http_session[i].socket = sock;
      tmp = tcp_sock_listen(sock,HTTP_PORT);
      if(tmp<0)
        reset_code(); //error, no se pudo poner a escuchar el socket
    }
```

Cuando se invoca a la función tcp_get_sock() los argumentos que se le pasan corresponden al tipo de socket (SERVER) y la función de HTTP que procesa los datos recibidos. En el tema 5.8.2 se explica esta función.

Abrir una sesión. Para abrir una sesión se utiliza la función **http_open_session(session)**, la cual recibe un identificador de sesión y cambia su estado a OPEN. No tiene valor de retorno. Esta función se invoca desde otra función de HTTP cuando un cliente se conecta y solicita un recurso.

Cerrar una sesión. Cuando un archivo asociado a una sesión se termina de transferir o se recibe una solicitud de desconexión del cliente, se invoca la función **http_close_session(session)**. Esta función recibe el identificador de la sesión que hay que cerrar y cambia su estado a CLOSED. Después llama a la función tcp_sock_listen() para que el socket asociado a la sesión continúe esperando nuevas conexiones.

Búsqueda de sesión. Cuando HTTP recibe datos provenientes de TCP, TCP envía a HTTP el número de socket que recibió la información y HTTP utiliza la función http_find_session() para encontrar una sesión asociada al socket. El argumento que recibe es el número de socket y regresa el identificador de sesión asociado. En caso de no encontrar ninguna sesión asociada el socket regresa un entero negativo. Esta función se invoca desde la función de recepción HTTP de la siguiente manera, donde *socket* y *session* son números enteros que identifican un socket y una sesión respectivamente:

```
session = http_find_session(socket);
```

5.8.2 Procesamiento de entrada de datos

HTTP implementa una función encargada de procesar la información recibida procedente del protocolo TCP. Esta función recibe el nombre de http_listener() y debe ser invocada por TCP a través del socket que recibe la información. Cuando se obtiene un socket TCP se le debe pasar el nombre de la función que procesa los datos de entrada, esta referencia se almacena en el socket y se invoca desde TCP de la siguiente manera, suponiendo que *socket* es una variable tipo struct tcp_sock:

```
socket.application = http_listener; //la referencia a la función se almacena al obtener el socket
/* invocación a la función de recepción HTTP desde la función  de recepción TCP  */
socket.application(sock_id, EVENT);
```

Los argumentos que recibe son un identificador de socket y un número entero que indica un evento, como puede ser la recepción de una solicitud de conexión, recepción de una confirmación de conexión, la recepción de datos, la recepción de una confirmación de datos, la recepción de una solicitud de desconexión, la recepción de un reset o la solicitud de un reenvío de datos.

A continuación se examina cada uno de los casos de recepción que se pueden presentar en esta función. Antes de evaluar a cuál de estos casos corresponde la recepción, http_listener() trata de asociar el socket que recibe con una sesión, invocando a la función http_find_session(), si no encuentra una sesión asociada al socket no procesa la información.

Solicitud de conexión. Este evento se genera cuando TCP recibe un paquete de solicitud de conexión con bandera de SYN mientras se encuentra en estado LISTENING. Antes de enviar una respuesta invoca a la función de recepción asociada al socket, si http_listener() puede asociar el socket a una sesión que espera recibir conexiones acepta la solicitud, le regresa un entero positivo a TCP y éste envía el segmento de confirmación de solicitud con el número de secuencia inicial. Si HTTP no pudiera asociar el socket con ninguna sesión, si por ejemplo ya todas las sesiones están en uso, regresa un entero negativo, lo cual le indica a TCP que no se puede procesar la solicitud de conexión y envía un segmento con RST.

Conexión establecida. Después de recibir una solicitud de conexión y pasar al estado SYN_RECEIVED, TCP espera la confirmación del número de secuencia inicial enviado para establecer la comunicación. Cuando TCP recibe la confirmación, finalizando las tres etapas del establecimiento de la conexión, invoca a la función de recepción de la aplicación HTTP, informándole el evento de *Conexión Establecida* y que a partir de este momento es posible enviar y recibir datos. Cuando http_listener() recibe este evento abre la sesión, invocando a la función http_open_session().

Recepción de datos. Después de establecerse la conexión entre el cliente y el servidor, el servidor recibe una solicitud de un recurso por parte del cliente, que debe contener la línea de solicitud con el formato que se analizó en el tema 4.11. Como este es el único paquete de datos que el servidor recibe del cliente, siempre que se reciben datos la función http_listener() verifica que comiencen con la palabra 'GET'. Si esto es así se busca en la solicitud el nombre del recurso, se aplica el algoritmo para obtener su *hash* y se recorre el arreglo de recursos buscando alguno que se identifique con el *hash* calculado. Al encontrar el recurso solicitado por el cliente, se guardan sus datos en la sesión encargada de transferirlo y se llama a la función de envío de datos, si el recurso no se encuentra, los datos de la página de error se almacenan en la sesión y si la solicitud no especifica un nombre de recurso, se almacenan los datos de la página de inicio.

Confirmación de datos. Después de recibir y procesar una solicitud de parte de un cliente, el flujo de la información será de servidor a cliente, transfiriendo el recurso solicitado quizá en varios paquetes, y esperando una confirmación del cliente por cada uno de ellos. Cuando se recibe una confirmación de datos, la función TCP que procesa el paquete invoca a la función de procesamiento de entrada de HTTP indicándole que se presenta el evento de *confirmación*. Para este evento HTTP ajusta los valores de la sesión: incrementa la variable que apunta al último byte enviado y asigna a la variable *datasent* el valor de cero.

Solicitud de desconexión. Si se llega a recibir una solicitud de desconexión por parte del cliente, se cierra la sesión llamando a la función http_close_session(). En este momento la sesión se libera y vuelve a estar disponible para ser usada por otro cliente.

Recepción de un paquete de reset. Se procede de la misma manera que en la solicitud de desconexión, se invoca a la función http_close_session() para que la sesión quede disponible para recibir nuevas conexiones.

Reenvío de datos. Cuando un paquete de datos enviado no fue recibido por el cliente es necesario el reenvío de los mismos. TCP indica a HTTP cuando esta situación ocurre llamando a la función de procesamiento de entrada con el evento correspondiente a *reenvío*. Para este caso simplemente se vuelve a enviar el paquete anterior, es por esto que el apuntador del archivo no se incrementa sino hasta haber recibido la confirmación, de esta manera es posible volver a formar el paquete de datos a partir del último byte confirmado.

La Figura 5.20 ilustra el algoritmo de la función de recepción http_listener().

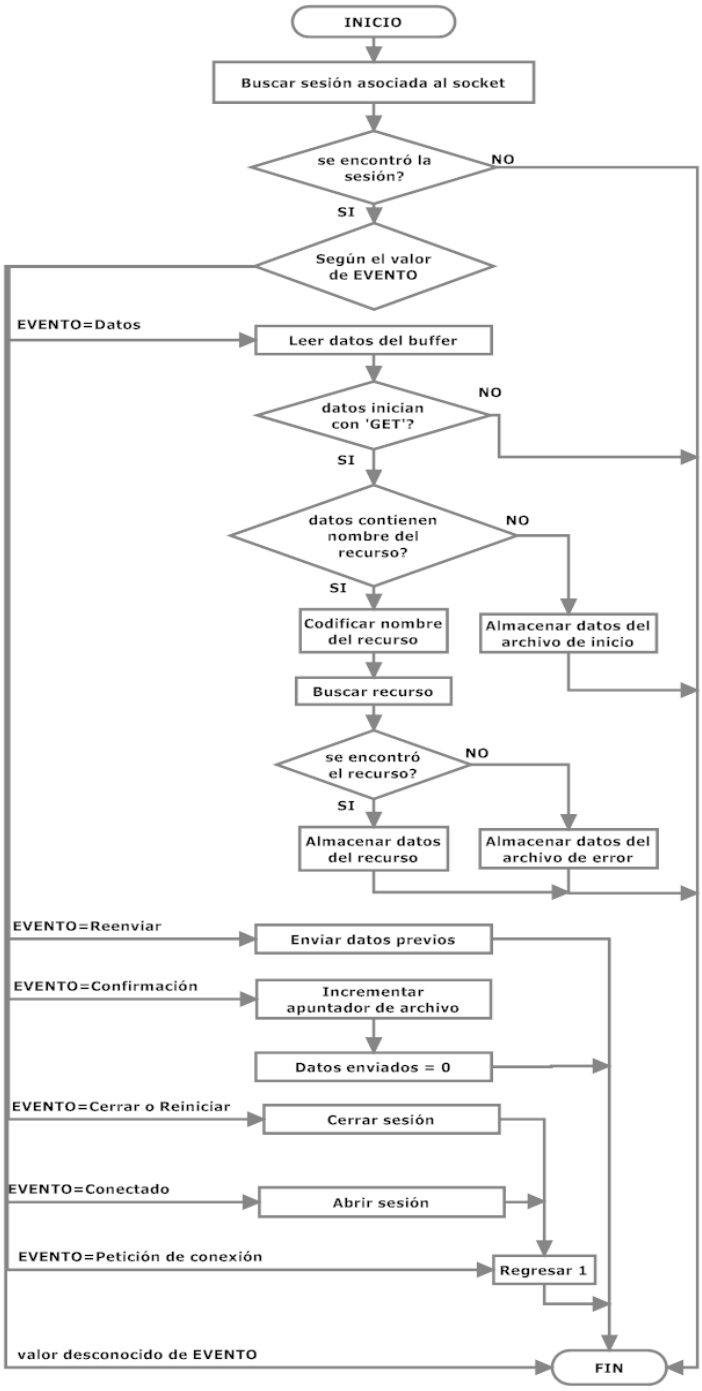


Figura 5.20 Función de recepción de la aplicación HTTP

5.8.3 Procesamiento de salida de datos

La función encargada de la transmisión de datos en HTTP recibe el nombre de http_send() se invoca desde la función periódica de HTTP. El argumento que recibe es el identificador de la sesión que tiene datos por enviar. No tiene valor de retorno. La forma de llamar a esta función es como se muestra a continuación, donde *session* es un número entero que identifica una sesión:

```
http_send(session);
```

La función valida los datos de la sesión, si no contiene una referencia a un archivo o la longitud del archivo es cero, no hay información por enviar y por lo tanto se sale de la función. Si el apuntador de archivo es igual a cero, indica que se va a transferir el inicio del archivo y por lo tanto hay que agregar la línea de estado al paquete que se está ensamblando, verificando la referencia al archivo, si contiene la página de error se escribe la línea "HTTP/1.0 404 Not found" y si contiene un identificador de archivo válido se escribe la línea "HTTP/1.0 200 OK". Después de esto se calcula la cantidad de datos que es posible enviar de acuerdo al tamaño de la ventana TCP y se termina de llenar un buffer con una cantidad de datos que no exceda ese tamaño de ventana. Una vez ensamblado el buffer con los datos de la aplicación se invoca a la función tcp_data_send() para enviarlos.

5.8.4 Función periódica HTTP

La función periódica del protocolo HTTP es el alma del servidor web, por eso se denomina **http_run();** no recibe argumentos ni regresa algún valor. Se invoca desde la función principal de manera periódica, después de haber habilitado el servicio mediante la función http_init_server(). De hecho estas son las dos únicas funciones de la capa de aplicación necesarias para implementar aplicaciones web. Primero se inicializa el servidor y después se invoca a la función http_run() de manera periódica (por ejemplo llamándola desde un ciclo infinito) y el servicio de transferencia de recursos web puede comenzar a utilizarse.

Cada vez que se hace una llamada a la función http_run() se hace un recorrido por cada una de las sesiones verificando si tienen datos pendientes por enviar, de ser así se invoca a la función http_send() para realizar el envío. Si se encuentra que una sesión ha terminado de transferir todo el archivo que almacena, se llama a la función http_close_session() para cerrar la sesión.

5.9 Implementación de una aplicación web

Haciendo uso de las funciones de la pila TCP/IP descritas en los párrafos anteriores es posible implementar aplicaciones web de manera muy sencilla. La idea principal es almacenar recursos en el servidor y acceder a ellos a través de un navegador web de una computadora para que su contenido sea visualizado en el monitor de la PC. En este tema se resumen algunos conceptos explicados en los temas anteriores para reforzar la comprensión sobre la implementación de una aplicación web.

Primero hay que configurar la dirección física y de red del microcontrolador usando la interfaz definida en el archivo *address.c* y realizar la conexión física del sistema a una red.

Los recursos o archivos web se programan utilizando cualquier herramienta, como editores HTML o simplemente en la aplicación de Windows conocida como *block de notas*. Una vez programada la página se utiliza algún programa para codificar archivos en ASCII. Si la página contiene imágenes u otros elementos éstos también se codifican y se almacenan en el servidor. Para almacenar los recursos en el servidor se abren los archivos *server.h* y *server.c*, en donde se encuentra la información de los recursos web. Es necesario calcular el *hash* de cada recurso web, utilizando un programa para esto en el cual se ingresa el nombre del recurso y se calcula el número correspondiente. Con estos datos, en el archivo *server.c* se declara una variable a la que se le asigna el contenido del recurso codificado y se añaden los datos del archivo al arreglo resources[]. El contenido de este documento se muestra en la Figura 5.21.

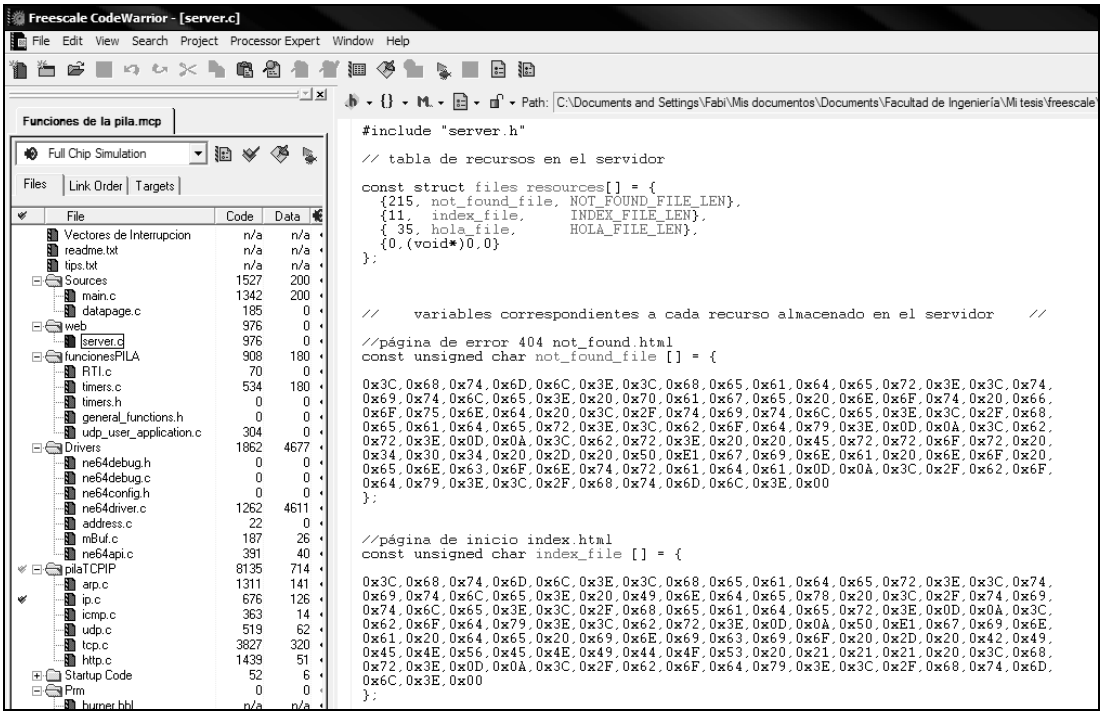


Figura 5.21 Cómo añadir recursos web a la memoria del microcontrolador

En el archivo *server.h* se añade una variable simbólica con la longitud del recurso. Este archivo contiene la declaración de variables externas (las que se encuentran declaradas en *server.c*) para que puedan ser usadas por el protocolo HTTP. El contenido del archivo se ilustra en la Figura 5.22.

En la función principal se invoca a las funciones de inicialización de todos los protocolos y módulos de la pila, posteriormente se invoca a las funciones periódicas dentro de un ciclo infinito. Con esto ya se tiene implementada una aplicación web en el microcontrolador que transfiere recursos a través de una red, mismos que pueden ser visualizados desde varias computadoras de manera simultánea. La Figura 5.23 muestra la programación de la aplicación.

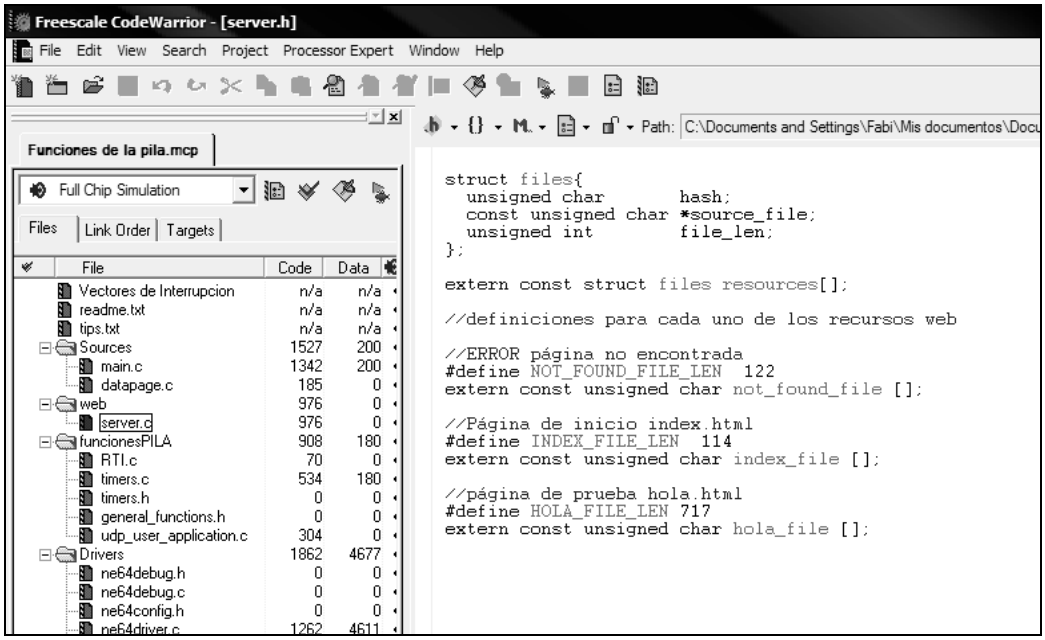


Figura 5.22 Contenido del archivo server.h

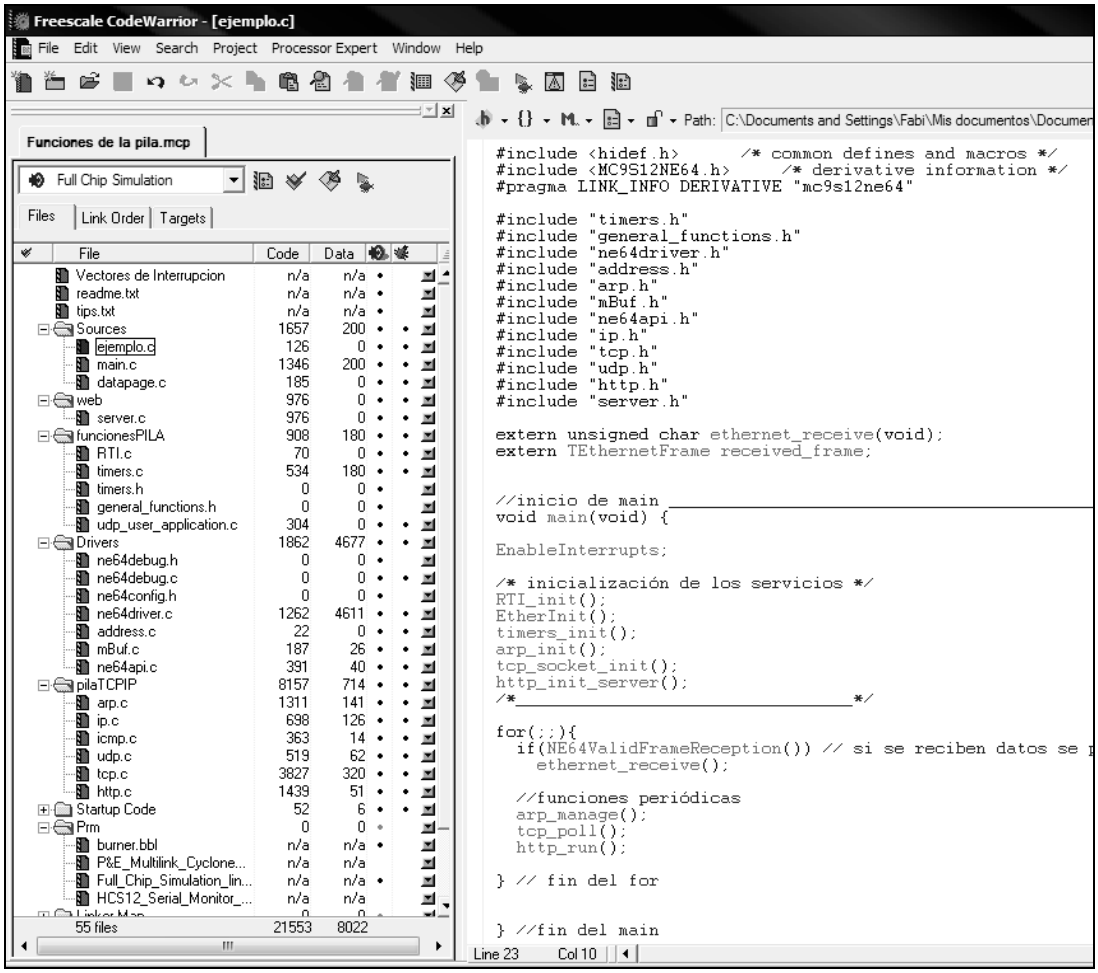


Figura 5.23 Ejemplo de programación de aplicación web