ÍNDICE

1 INTRODUCCIÓN

1.1 OBJETIVOS

En este proyecto se va a implementar una pila TCP/IP para un microcontrolador MMC2107. Se diseñarán todas las funciones necesarias para ello creando una librería para que cualquier usuario pueda establecer una comunicación entre dos microcontroladores, entre un microcontrolador y un PC, entre dos PCs…

Los protocolos que se implementarán serán TCP e IP ya que éstos permiten comunicar dos extremos, encaminando los paquetes de uno a otro de tal forma que resulte una comunicación fiable.

Se va a proceder a explicar más detenidamente qué es un modelo estratificado de protocolos y por qué se opta por implementar la pila TCP/IP frente a otros protocolos.

1.2 MODELO ESTRATIFICADO DE LOS PROTOCOLOS

Para poder entender una forma de comunicación sin conocer los detalles del hardware se encuentran los protocolos. Éstos contienen los detalles referentes a los formatos de los mensajes, describen cómo se responde cuando llega un mensaje y especifican cómo se manejan los errores u otras condiciones anormales.

Los sistemas complejos de comunicación de datos no utilizan un solo protocolo para manejar todas las tareas de transmisión, sino que requieren de un conjunto de protocolos cooperativos.

Este conjunto de protocolos adopta un modelo estratificado para realizar las diferentes funciones encomendadas. Se tiene que tomar una decisión a cerca del número de capas óptimo. Para ello se toma el modelo presentado por ISO (International Organization for Standardization) se denomina OSI (Open System Interconnection). Éste contiene 7 niveles bien diferenciados como se representa a continuación:

Figura 1.1

Existe un segundo modelo de estratificación, el cual se presenta a continuación:

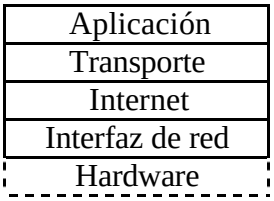

Figura 1.2

El nivel de aplicación es el más alto. Se encarga de enviar los datos proporcionados por el usuario al nivel que se encuentra por debajo para enviarlos al otro extremo de la conexión o entregarle los que lleguen del host remoto. Una aplicación interactúa con uno de los protocolos de nivel de transporte para enviar o recibir los datos. Cada programa de aplicación selecciona el tipo de transporte necesario, el cual puede ser una secuencia de mensajes individuales o un flujo continuo de octetos.

El objetivo principal del nivel de transporte es el de proporcionar la comunicación entre dos entidades regulando el flujo de información. Puede proporcionar un transporte fiable, asegurando que los datos lleguen sin errores. En este nivel se encuentran los protocolos TCP o UDP. Habrá que elegir el que proporcione una mayor seguridad en el transporte de la información.

La capa de Internet maneja la comunicación de una máquina con otra. Ésta acepta una solicitud para enviar un paquete desde la capa de transporte, junto con un identificador de máquina. En este nivel se encuentra el protocolo IP.

La interfaz de red es el responsable de aceptar los datagramas IP y transmitirlos hacia una red específica. Aquí se encontrará el protocolo PPP.

En la siguiente figura se encuentra una posible jerarquía de protocolos correspondientes a esta estratificación. Se estudiará qué protocolos son los más adecuados para el objetivo del proyecto.

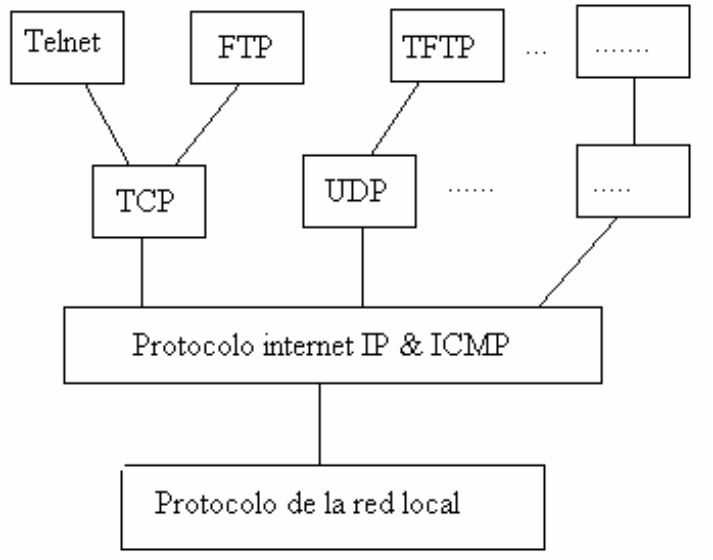

Figura 1.3

1.2.1 PROTOCOLO IP

El protocolo IP (Internet Protocol) es un protocolo de red (protocolo de nivel 3) y será uno de los protocolos que se tendrán que implementar.

Está diseñado para interconectar sistemas basados en redes de intercambio de paquetes. Este protocolo permite el intercambio de bloque de datos, llamados datagramas, desde un host origen a uno destino. Cada uno de estos host se identifica con una dirección IP de longitud fija.

El protocolo IP, además de encaminar un datagrama por la red utilizando la dirección IP del host destino, también se encarga, si es necesario y está configurado para ello, de fragmentar y reensamblar grandes datagramas para su transmisión a través de redes más pequeñas.

El protocolo IP trata cada datagrama como una unidad independiente del resto de los datagramas.

El protocolo Internet no proporciona ningún mecanismo de comunicación fiable. No existen acuses de recibo entre extremos. No hay control de errores para los datos, sólo una suma de control de la cabecera. No hay retransmisiones y no existe control de flujos.

Este protocolo interactúa por un lado con el protocolo de la red local PPP, y por otro con UDP o TCP. Se deberá elegir uno de estos dos protocolos para que junto con IP, creemos la torre de protocolos para la comunicación. Para ello se va a proceder a introducir las características más importante de cada uno para poder elegir el más adecuado.

1.2.2 PROTOCOLOS DE TRANSPORTE

1.2.2.1 Protocolo UDP

El Protocolo de Datagrama de Usuario o UDP (User Datagram Protocol) es un protocolo muy simple. Proporciona el mecanismo básico para enviar datagramas desde un programa de nivel de aplicación a otros programas del mismo nivel.

Este protocolo es NO orientado a conexión, sólo transporta datagramas (Flujo de bytes) desde una máquina origen a otra destino. El servicio que ofrece UDP es NO fiable ya que no añade nada nuevo a IP y éste no lo es. En los problemas de duplicación, pérdidas, retardos, …, UDP no hace nada al respecto. Por esto UDP suele utilizarse en entornos de redes de área local donde las redes son fiables y no es necesario un control riguroso del tráfico de datos.

UDP llevará unas direcciones de extremo a extremo para distinguir usuarios o aplicaciones en una máquina. En vez de denominarse direcciones como en IP, reciben en nombre de *puertos.*

1.2.2.2 Protocolo TCP

El Protocolo de Control de Transmisión o TCP (Transmisión Control Protocol) presta un servicio fiable y en modo conectado, por ello es el método más eficiente y seguro de mover tráfico de red entre un cliente y un servidor.

TCP permite el funcionamiento en full-duplex y utiliza una técnica conocida como *acuse de recibo* para garantizar la llegada de los datos a la entidad remota. Cada vez que envía un mensaje pone en marcha un temporizador. Si éste expira sin que se haya recibido el paquete de asentimiento se reenviará dicho mensaje.

En TCP también existe el concepto de *puerto,* pero es más complejo que en UDP, donde el puerto representa al usuario. Aquí se define el usuario o extremo por la pareja (dirección IP, puerto) y la conexión queda definida por los dos extremos. Por tanto, es posible compartir un mismo puerto por varias conexiones.

1.2.2.3 Comparación entre UDP y TCP

Tras estudiar las características de cada protocolo, se observa que TCP es mejor frente UDP ya que nos proporciona una conexión más fiable con recuperación de errores y de datos, si el mensaje se perdiera en la red o no llegara en su orden. Así pues, en el proyecto se implementará TCP sobre IP.

1.2.3 OTROS PROTOCOLOS

Para que cualquier dispositivo interactúe con una red de datos, éste debe estar provisto de programas que establezcan la conexión a la red, es decir, programas que conviertan la información de las capas de nivel superior en tramas que puedan viajar por la red.

1.2.3.1 Protocolo punto a punto (PPP)

PPP está definido dentro de la familia de protocolos denominados TCP/IP y se ubica en los niveles bajos de esta familia permitiendo que la computadora o placa en la que esté programado, se pueda comunicar con la red.

PPP permite el intercambio de datagramas entre dos host a través de un enlace de comunicaciones. Dicho enlace debe ofrecer una comunicación fux_duplex y un transporte ordenado de los datagramas.

1.2.3.2 Ethernet

Ethernet suele utilizarse para referirse a todas las LANs tipo "carrier sense multiple access/collision detection (CSMA/CD)". Ethernet está bien adaptada a las aplicaciones en

que el soporte de comunicaciones local a menudo tiene que procesar un elevado tráfico con puntas elevadas de intercambio de datos.

Ethernet se basa en redes de difusión, lo que significa que todas las estaciones ven todos los paquetes, sin tener en cuenta si representan un destino determinado. Cada estación debe examinar los paquetes recibidos para determinar si la estación es un destino. En este caso, el paquete se pasa a una capa de protocolo superior para su procesamiento adecuado.

Ethernet proporciona servicios correspondientes a las capas 1 y 2 del modelo de referencia OSI. Está implementada en hardware, en general a través de una tarjeta de interface en un ordenador o a través de una placa principal en el propio ordenador.

1.3 MICROCONTROLADOR MMC2107

El microcontrolador elegido es el MMC2107, primer miembro de la familia de micros basados en la CPU MCORE M210.

1.4 ESTRUCTURACIÓN DE LA MEMORIA DEL PROYECTO

La memoria se dividirá en varias partes. En primer lugar se describirán las características del MMC2170 así como el funcionamiento de los periféricos necesarios para este proyecto. Entre ellos se encuentran los temporizadores (PIT) y el puerto serie asíncrono (SCI). Se estudiarán los diagramas de bloques, mapas de memoria, modo de operación…

Una vez descrito el microcontrolador se pasará a estudiar detenidamente el funcionamiento de los protocolos TCP e IP: formato de las cabeceras, control de flujos, modo de operación, diagrama de estados de una conexión… También se comentarán algunos protocolos de nivel físico como PPP.

En el siguiente punto se conocerá el funcionamiento de cada una de las funciones que se implementan en el proyecto, tanto las que implementan los protocolos como las específicas del microcontrolador. Se conocerán los parámetros que necesitan, los valores que retornan, los errores que se pueden producir… en definitiva, se intentará explicar cada una de las funciones internamente, con detalle.

Para saber como funciona, y sobre todo que funciona, se añadirá un ejemplo y un protocolo de pruebas donde comprobar que el programa es correcto y hace todo lo que necesita TCP/IP para comunicar dos extremos.

Se comentarán algunas limitaciones que existen en este proyecto así como algunas aplicaciones prácticas de la pila TCP/IP.

En el apartado de anexos se adjuntan varios puntos. El primero de ellos es un manual de usuario para que el lector que quiera establecer la implementación de la pila TCP/IP conozca qué funciones son las que necesita sin tener que entrar a saber cómo están

realizadas internamente. En este manual sólo aparecerán las funciones de más alto nivel, las que el usuario debe utilizar, explicando para que sirven y los parámetros que se le deben pasar.

En el segundo punto se hará una descripción de las placas utilizadas en el proyecto asó como la conexión entre ellas.

En un tercer punto aparece descrito el puerto serie del PC, registros que utiliza, funcionamiento, …, ya que es necesaria su utilización para poder realizar el protocolo de prueba.

Por último se adjuntará el código C de todas las funciones.

2 MICROCONTROLADOR MMC2107

2.1 DESCRIPCIÓN GENERAL

2.1.1 INTRODUCCIÓN

El microcontrolador MMC2107 es el primer miembro de una familia de microcontroladores basados en la CPU MCORE M210. Opera con voltajes entre 2.7 voltios y 3.6. La frecuencia máxima de operación es de 33MHz y el rango de temperatura sobre el que trabaja es de –40º a 85º.

Se puede encontrar en dos encapsulados, uno con 100 pines en el que no se puede acceder a memoria externa, y otro de 144 pines que se usará cuando es necesario utilizar memoria externa. Los pines que no están disponibles en el encapsulado de 100 se observan en la siguiente tabla:

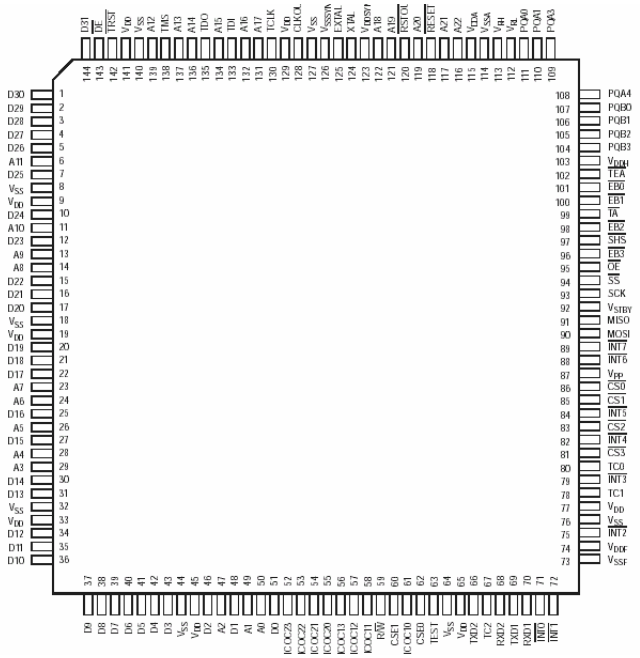

Figura 2.1

External Memory interface and Ports								
D[31:0]	PA[7:0], PB[7:0] PC[7:0], PD[7:0]	32	I/O	Y	Y	LOAD	—	ST
$\overline{SHS}$	$\overline{RCON}$ / PE7	1	I/O	Y	Y	LOAD	Pullup	ST
$\overline{TA}$	PE6	1	I/O	Y	Y	LOAD	Pullup	ST
TEA	PE5	1	I/O	Y	Y	LOAD	Pullup	ST
CSE[1:0]	PE[4:3]	2	I/O	Y	Y	LOAD	Pullup	ST
TC[2:0]	PE[2:0]	3	I/O	Y	Y	LOAD	Pullup	ST
R/$\overline{W}$	PF7	1	I/O	Y	Y	LOAD	Pullup	ST
A[22:0]	PF[6:0], PG[7:0] PH[7:0]	23	I/O	Y	Y	LOAD	Pullup	ST
$\overline{EB}$[3:0]	PI[7:4]	4	I/O	Y	Y	LOAD	Pullup	ST
$\overline{CS}$[3:0]	PI[3:0]	4	I/O	Y	Y	LOAD	Pullup	ST
$\overline{OE}$	—	1	I/O(6)	—	—	LOAD	—	ST

Tabla 2.1

2.1.2 CARACTERÍSTICAS

Las características del MMC2107 junto con la de algunos periféricos son:

- Procesador MCORE M210:
  - Arquitectura RISC, 32 bits.
  - Bajo consumo y alta ejecución.
- FLASH 128 kbyte:
  - Para programar y borrar la FLASH se requiere V_{pp} externa a 5 voltios.
  - Tamaño de bloque 16 K.
  - Longitud de la palabra 32 bits.
- SRAM de 8 kbytes.
  - Byte, half-word(16 bits) y word (32 bits) acceso read/write.
  - No se puede acceder en modo de bajo consumo.
- Puerto serie síncrono (SPI):
  - Modo maestro/esclavo.
  - Reloj serie con polaridad y fase programable.
  - Modo Wired-OR.
- Dos módulos serie de comunicación (SCI):
  - Operación full-duplex.
  - Formato de dato programable a 8 ó 9 bits.
- Dos temporizadores:
  - Arquitectura 16 bits.

  - Acumulador de 16 bits.
  - Preescaler.
- Convertidor analógico-digital:
  - 8 entradas analógicas.
  - Varios modos de muestreo.
  - Mantenimiento y muestreo interno.
  - Dos colas para los datos digitalizados.
- Controlador de interrupciones:
  - Hasta 40 fuentes de interrupciones.
  - Interrupciones urgentes y normales.
  - Máscara para un determinado nivel de prioridad.
- Interrupciones externas.
  - 8 entradas para captura de interrupciones externas.
  - Puede ser por flanco o nivel.
- Watchdog:
  - Evitar que el micro se quede en un bucle infinito.
  - Contador de 16 bits.
- PLL (Phase-lock loop):
  - Cristal de referencia desde 2 a 10 MHz.
- Reset:
  - Separa la señal de reset de salida y de entrada.
  - Se puede conocer la causa del reset vía software.
  - 6 fuentes de reset
    - Reset externo
    - Perdida de alimentación
    - Watchdog
    - PLL pierde la fase
    - PLL pierde el reloj
    - Software

2.1.3 DIAGRAMA DE BLOQUES

La estructura básica del MMC2107 se muestra en la siguiente figura:

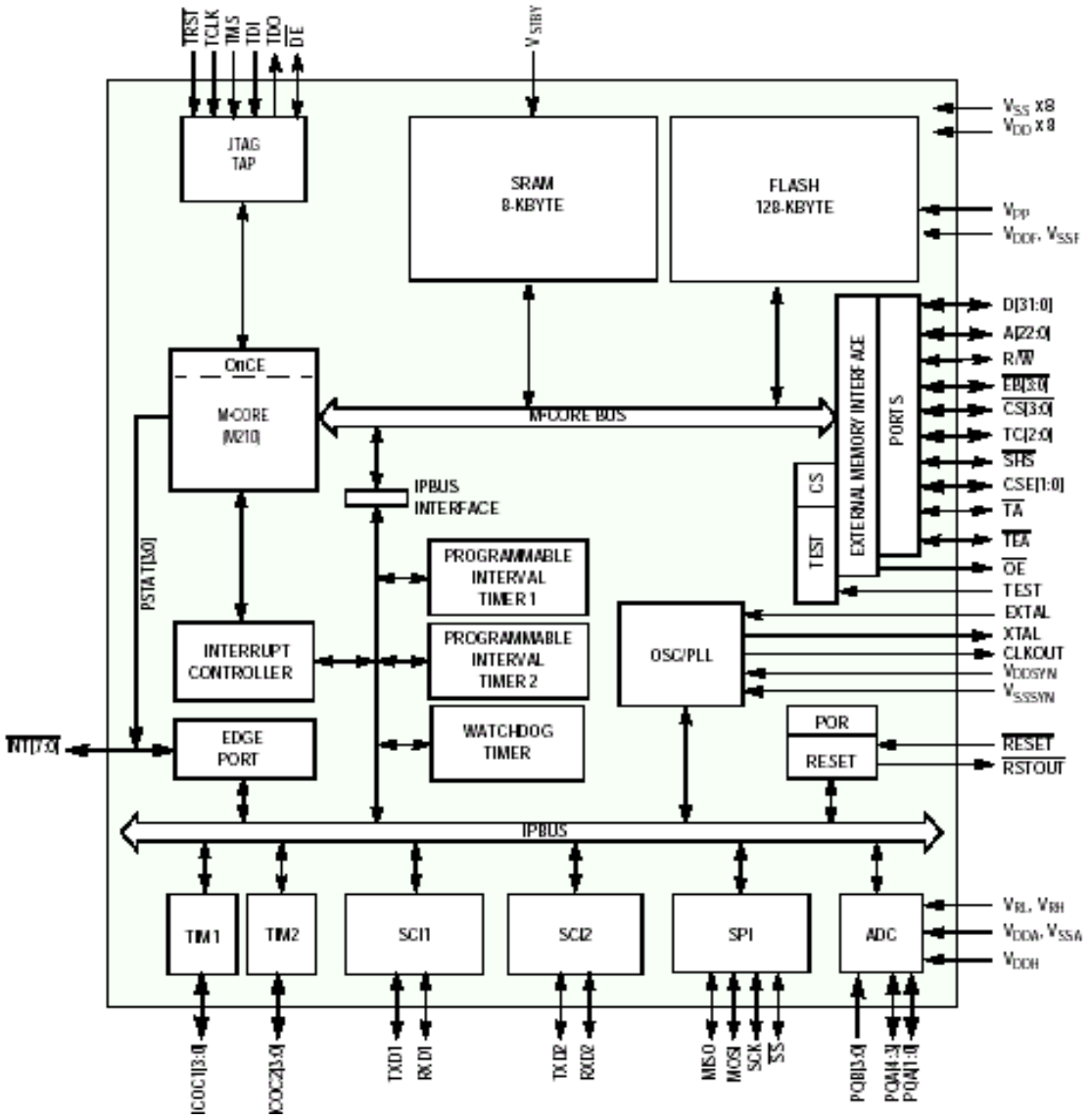

Figura 2.2

2.1.4 MAPA DE MEMORIA

El mapa de memoria es el siguiente:

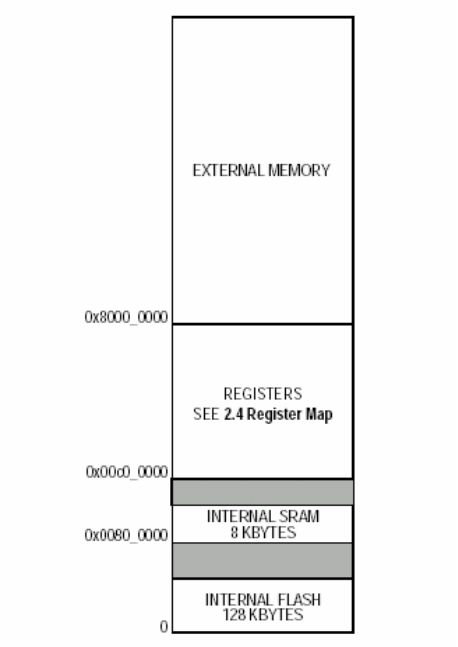

Figura 2.3

La siguiente tabla informa de las direcciones de los registros asociados a los periféricos:

Base Address (Hex)	Usage
0x00c0_0000	Ports[2] (PORTS)
0x00c1_0000	Chip configuration (CCM)
0x00c2_0000	Chip selects (CS)
0x00c3_0000	Clocks (CLOCK)
0x00c4_0000	Reset (RESET)
0x00c5_0000	Interrupt controller (INTC)
0x00c6_0000	Edge port (EPORT)
0x00c7_0000	Watchdog timer (WDT)
0x00c8_0000	Programmable interrupt timer 1 (PIT1)
0x00c9_0000	Programmable interrupt timer 2 (PIT2)
0x00ca_0000	Queued analog-to-digital converter (QADC)
0x00cb_0000	Serial peripheral interface (SPI)
0x00cc_0000	Serial communications interface 1 (SCI1)
0x00cd_0000	Serial communications interface 2 (SCI2)
0x00ce_0000	Timer 1 (TIM1)
0x00cf_0000	Timer 2 (TIM2)
0x00d0_0000	FLASH registers (CMFR)

Tabla 2.2

2.1.5 MÓDULO DE CONFIGURACIÓN (CCM)

Este módulo controla la configuración y el modo de operación, es decir, permite seleccionar el modo de funcionamiento. Las características son las siguientes:

- Selección del modo de operación:
  - Master mode
  - Single-chip mode
  - Emulation mode
  - Factory access slave test
- Modo de obtención de reloj: modo con referencia externa o interna.
- Selecciona el dispositivo de arranque.
- Selecciona si la FLASH está habilitada o no.

La configuración por defecto es la siguiente:

- Single mode.
- Modo normal de funcionamiento del PLL.
- La referencia del PLL es un cristal.
- Flash habilitada.
- Arranque desde dispositivo interno.

2.2 MÓDULO CONTROLADOR DEL RESET

2.2.1 CARACTERÍSTICAS

- Existen 6 fuentes de reset:
  - Externo
  - Pérdida de alimentación
  - Watchdog
  - PLL pierde la fase
  - PLL pierde el reloj
  - Software
- El pin RSTOUT negado estará activo durante 512 ciclos después del reset.

2.2.2 DIAGRAMA DE BLOQUES

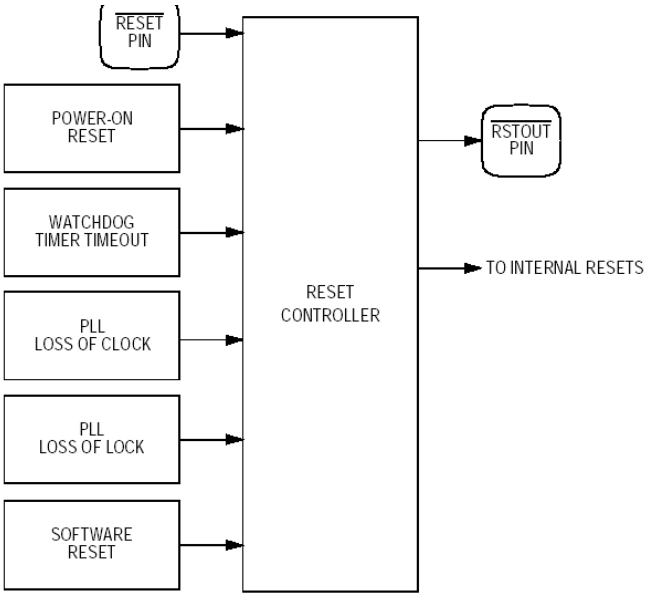

Figura 2.4

2.2.3 MAPA DE MEMORIA

Address	Bits 7–0	Access(1)
0x000c4_0000	Reset control register (RCR)	S/U
0x000c4_0001	Reset status register (RSR)	S/U
0x000c4_0002	Reset test register (RTR)	S/U
0x000c4_0003	Reserved(2)	—

Tabla 2.3

2.3 MÓDULO CONTROLADOR DE INTERRUPCIONES

2.3.1 INTRODUCCIÓN

El controlador de interrupciones recoge la solicitud de múltiples fuentes de interrupción.

2.3.2 CARACTERÍSTICAS

Algunas de las características principales de este módulo son las siguientes:

- 40 fuentes de interrupción.
- 32 niveles de prioridad programable para cada una de las fuentes de interrupción.
- Selección de solicitud de interrupción normal o rápida para cada nivel de prioridad.
- Generación de vector de interrupciones basado en el nivel de prioridad.

2.3.3 DESCRIPCIÓN FUNCIONAL

Cada fuente de interrupción envía una única señal al controlador de interrupciones. Cada una de ellas se puede programar con uno de los 32 niveles de prioridad usando el registro PLSR. La prioridad mayor es el nivel 31 y la menor el nivel 0.

Pueden existir interrupciones normales o rápidas. Estas últimas siempre tienen prioridad respecto a las interrupciones normales, incluso si el nivel de prioridad de la interrupción normal es superior al nivel de prioridad de la rápida.

Si la señal de interrupción rápida llega cuando la señal de una interrupción normal ya ha llegado, la señal de interrupción normal será anulada.

El controlador de interrupciones asigna un número a cada una de las fuentes de interrupciones, como se observa en la siguiente tabla:

Source	Module	Flag	Source Description	Flag Clearing Mechanism
16	TIM1	C0F	Timer channel 0	Write C0F = 1 or access IC/OC if TFFCA = 1
17		C1F	Timer channel 1	Write 1 to C1F or access IC/OC if TFFCA = 1
18		C2F	Timer channel 2	Write 1 to C2F or access IC/OC if TFFCA = 1
19		C3F	Timer channel 3	Write 1 to C3F or access IC/OC if TFFCA = 1
20		TOF	Timer overflow	Write TOF = 1 or access TIMCNTH/L if TFFCA = 1
21		PAIF	Pulse accumulator input	Write PAIF = 1 or access PAC if TFFCA = 1
22		PAOVF	Pulse accumulator overflow	Write PAOVF = 1 or access PAC if TFFCA = 1
23	TIM2	C0F	Timer channel 0	Write C0F = 1 or access IC/OC if TFFCA = 1
24		C1F	Timer channel 1	Write C1F = 1 or access IC/OC if TFFCA = 1
25		C2F	Timer channel 2	Write C2F = 1 or access IC/OC if TFFCA = 1
26		C3F	Timer channel 3	Write C3F = 1 or access IC/OC if TFFCA = 1
27		TOF	Timer overflow	Write TOF = 1 or access TIMCNTH/L if TFFCA = 1
28		PAIF	Pulse accumulator input	Write PAIF = 1 or access PAC if TFFCA = 1
29		PAOVF	Pulse accumulator overflow	Write PAOVF = 1 or access PAC if TFFCA = 1
30	PIT1	PIF	PIT interrupt flag	Write PIF = 1 or write PMR
31	PIT2	PIF	PIT interrupt flag	Write PIF = 1 or write PMR
32	EPORT	EPF0	Edge port flag 0	Write EPF0 = 1
33		EPF1	Edge port flag 1	Write EPF1 = 1
34		EPF2	Edge port flag 2	Write EPF2 = 1
35		EPF3	Edge port flag 3	Write EPF3 = 1
36		EPF4	Edge port flag 4	Write EPF4 = 1
37		EPF5	Edge port flag 5	Write EPF5 = 1
38		EPF6	Edge port flag 6	Write EPF6 = 1
39		EPF7	Edge port flag 7	Write EPF7 = 1

Tabla 2.4

2.3.4 DIAGRAMA DE BLOQUES

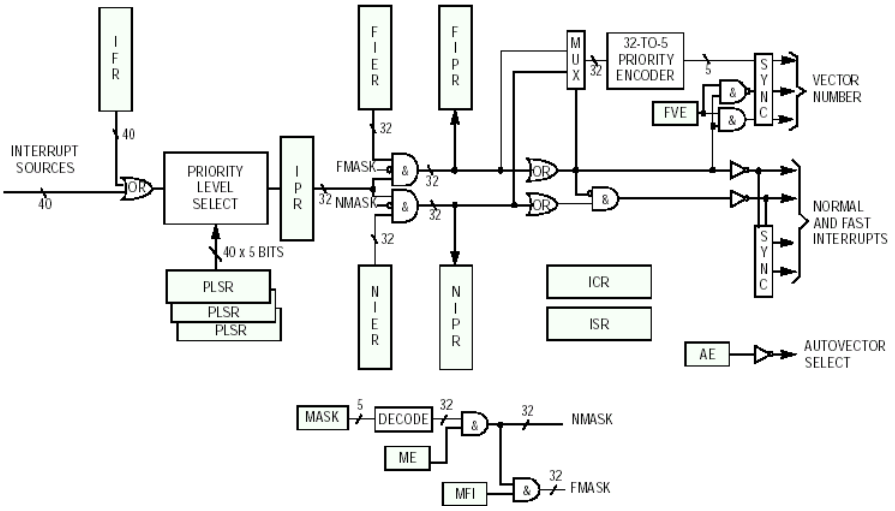

Figura 2.5

2.3.5 MAPA DE MEMORIA

Address	Bits 31–24	Bits 23–16	Bits 15–8	Bits 7–0	Access[1]
0x00c5_0000	Interrupt control register (ICR)		Interrupt status register (ISR)		S/U
0x00c5_0004	Interrupt force register high (IFRH)				S/U
0x00c5_0008	Interrupt force register low (IFRL)				S/U
0x00c5_000c	Interrupt pending register (IPR)				S/U
0x00c5_0010	Normal interrupt enable register (NIER)				S/U
0x00c5_0014	Normal interrupt pending register (NIPR)				S/U
0x00c5_0018	Fast interrupt enable register (FIER)				S/U
0x00c5_001c	Fast interrupt pending register (FIPR)				S/U
0x00c5_0020 through 0x00c5_003c	Unimplemented[2]				—
Priority level select registers (PLSR0–PLSR39)					
0x00c5_0040	PLSR0	PLSR1	PLSR2	PLSR3	S
0x00c5_0044	PLSR4	PLSR5	PLSR6	PLSR7	S
0x00c5_0048	PLSR8	PLSR9	PLSR10	PLSR11	S
0x00c5_004c	PLSR12	PLSR13	PLSR14	PLSR15	S
0x00c5_0050	PLSR16	PLSR17	PLSR18	PLSR19	S
0x00c5_0054	PLSR20	PLSR21	PLSR22	PLSR23	S
0x00c5_0058	PLSR24	PLSR25	PLSR26	PLSR27	S
0x00c5_005c	PLSR28	PLSR29	PLSR30	PLSR31	S
0x00c5_0060	PLSR32	PLSR33	PLSR34	PLSR35	S
0x00c5_0064	PLSR36	PLSR37	PLSR38	PLSR39	S
0x00c5_0068 through 0x00c5_007c	Unimplemented[2]				—

Tabla 2.5

2.4 TEMPORIZADOR (TIM1 Y TIM2)

2.4.1 INTRODUCCIÓN

El MMC2107 tiene dos módulos temporizadores de cuatro canales. Cada uno de ellos consta de un contador programable de 16-bit por un preescaler programable 7-stage. Cada uno de los cuatro canales puede ser configurado como entrada o salida. Adicionalmente, uno de los canales puede ser configurado como un pulso acumulador.

2.4.2 CARACTERÍSTICAS

Algunas de las características de los temporizadores son:

- Cuarto canales de 16-bit
- Arquitectura 16-bit
- Preescaler programable.
- Pulso acumulador de 16-bit

2.4.3 DESCRIPCIÓN FUNCIONAL

El módulo temporizador es un contador de 16 bits y cuatro canales que puede ser utilizado como temporizador de entrada, como generador de distintas señales de ondas en función de una comparación de salida y como pulso acumulador. Se dispondrá de 8 canales, cada 4 de ellos está asociado a un contador.

2.4.3.1 Preescaler

El preescaler divide el reloj por 1, 2, 4, 8, 16, 32, 64 o 128. Los bits PR[2:0] en el registro TIMSCR2 seleccionan el valor del divisor.

2.4.3.2 Temporizador de entrada

Escribiendo un cero en el bit IOSx, se configura el canal x como un canal de captura de entrada. Esta función captura el contador cuando ocurre un evento externo.

La mínima anchura de un pulso para la captura a la entrada debe ser mayor de dos módulos de reloj.

2.4.3.3 Generación de señal de salida

Seleccionando el bit IOSx (escribiendo un uno) se configura el canal x como canal de comparación de salida. Esta función puede generar un pulso periódico con una polaridad, duración y frecuencia programable. Cuando el contador llega al valor que hay en el registro del canal tras una comparación de un canal de salida, el contador puede fijar, limpiar… el pin del canal.

2.4.3.4 Pulso acumulador

El pulso acumulador (PA) es un contador de 16 bits que puede operar en dos modos:

- Modo contador de eventos: Cuenta cambios en los flancos de una determinada polaridad en un pin de entrada (PAI) que tendrá un pulso acumulador.
- Modo contador acumulador: Cuenta pulsos desde un reloj *divide-by-64*.

2.4.3.5 Propósito general de los puertos I/O

Los temporizadores pueden ser configurados para realizar algunas de las funciones anteriormente mencionadas. Para ello se tendrá que realizar lo siguiente:

- Para configurar un pin para una captura de entrada:
  - Limpiar el bit IOS en el registro TIMIOS.
  - Limpiar EL bit DDR en TIMDDR.
  - Escribir en TIMCTL2 para seleccionar el flanco que quiere detectar.

- Para configurar el pin para hacer una comparación de salida:
  - Seleccionar el bit IOS en el registro TIMIOS.
  - Escribir EL valor de la comparación de salida en TIMCxH/L.
  - Limpiar el bit DDR en TIMDDR.
  - Escribir en el bit Omx/Olx en TIMCTL1 para seleccionar la acción de salida.

2.4.4 DIAGRAMA DE BLOQUES

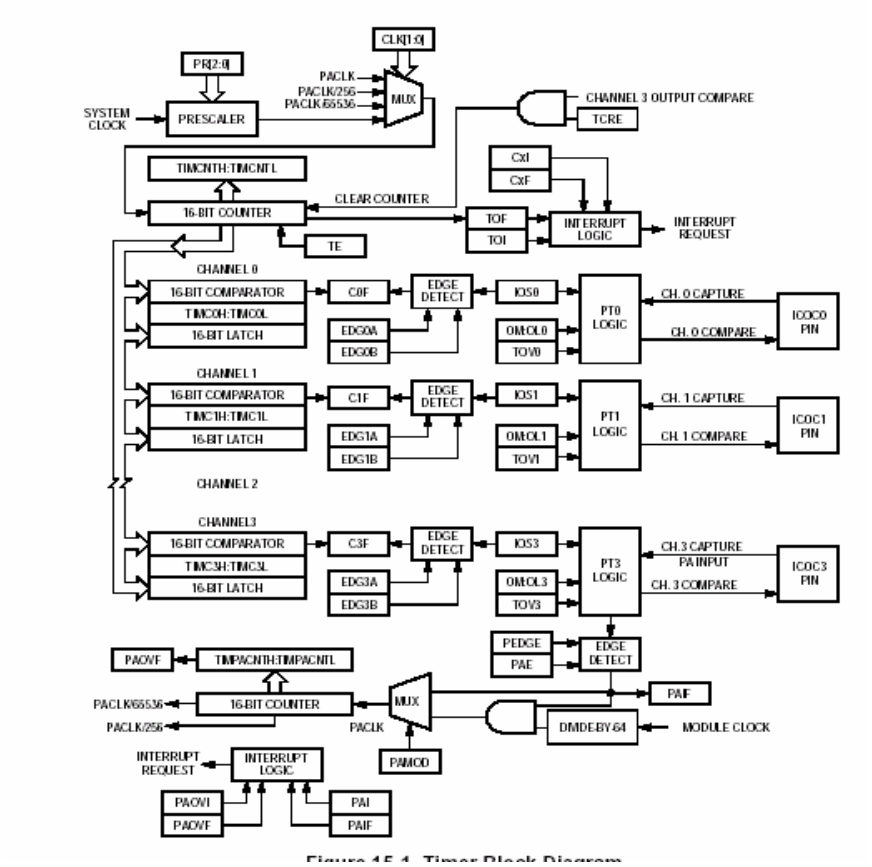

Figura 2.6

2.4.5 MAPA DE MEMORIA

En la siguiente tabla se puede observar el mapa de memoria de los dos módulos temporizadores. El temporizador 1 tiene una dirección base de 0x00ce_0000 y el temporizador 2 la tiene en 0x00cf_0000.

Address		Bits 7–0	Access[1]
TIM1	TIM2		
0x00ce_0000	0x00cf_0000	Timer IC/OC select register (TIMIOS)	S
0x00ce_0001	0x00cf_0001	Timer compare force register (TIMCFORC)	S
0x00ce_0002	0x00cf_0002	Timer output compare 3 mask register (TIMOC3M)	S
0x00ce_0003	0x00cf_0003	Timer output compare 3 data register (TIMOC3D)	S
0x00ce_0004	0x00cf_0004	Timer counter register high (TIMCNTH)	S
0x00ce_0005	0x00cf_0005	Timer counter register low (TIMCNTL)	S
0x00ce_0006	0x00cf_0006	Timer system control register 1 (TIMSCR1)	S
0x00ce_0007	0x00cf_0007	Reserved[2]	—
0x00ce_0008	0x00cf_0008	Timer toggle-on-overflow register (TMTOV)	S
0x00ce_0009	0x00cf_0009	Timer control register 1 (TIMCTL1)	S
0x00ce_000a	0x00cf_000a	Reserved[2]	—
0x00ce_000b	0x00cf_000b	Timer control register 2 (TIMCTL2)	S
0x00ce_000c	0x00cf_000c	Timer interrupt enable register (TIMIE)	S
0x00ce_000d	0x00cf_000d	Timer system control register 2 (TIMSCR2)	S
0x00ce_000e	0x00cf_000e	Timer flag register 1 (TIMFLG1)	S
0x00ce_000f	0x00cf_000f	Timer flag register 2 (TIMFLG2)	S
0x00ce_0010	0x00cf_0010	Timer channel 0 register high (TIMC0H)	S
0x00ce_0011	0x00cf_0011	Timer channel 0 register low (TIMC0L)	S
0x00ce_0012	0x00cf_0012	Timer channel 1 register high (TIMC1H)	S
0x00ce_0013	0x00cf_0013	Timer channel 1 register low (TIMC1L)	S
0x00ce_0014	0x00cf_0014	Timer channel 2 register high (TIMC2H)	S
0x00ce_0015	0x00cf_0015	Timer channel 2 register low (TIMC2L)	S
0x00ce_0016	0x00cf_0016	Timer channel 3 register high (TIMC3H)	S
0x00ce_0017	0x00cf_0017	Timer channel 3 register low (TIMC3L)	S
0x00ce_0018	0x00cf_0018	Pulse accumulator control register (TIMPACTL)	S
0x00ce_0019	0x00cf_0019	Pulse accumulator flag register (TIMPAFLG)	S
0x00ce_001a	0x00cf_001a	Pulse accumulator counter register high (TIMPACNTH)	S
0x00ce_001b	0x00cf_001b	Pulse accumulator counter register low (TIMPACNTL)	S
0x00ce_001c	0x00cf_001c	Reserved[2]	—
0x00ce_001d	0x00cf_001d	Timer port data register (TIMPORT)	S
0x00ce_001e	0x00cf_001e	Timer port data direction register (TIMDDR)	S
0x00ce_001f	0x00cf_001f	Timer test register (TIMTST)	S

Tabla 2.6

2.5 MÓDULO SERIE DE COMUNICACIONES (SCI1 Y SCI2)

2.5.1 INTRODUCCIÓN

El microcontrolador dispone de dos puertos serie asíncronos, cada uno con sus propios registros de control y sus pines de entrada/salida (I/O).

El SCI transmite y recibe datos independientemente.

Se hablará genéricamente tanto del SCI1 como del SCI2 denotándolo SCI, al igual que se llamará SCIPORT para referirse tanto a SCI1PORT como a SCI2PORT.

2.5.2 CARACTERÍSTICAS

Algunas de características de cada uno de los puertos serie síncrono son las siguientes:

- Operación full-duplex.
- Formato estándar NRZ (non-return-to-zero).

- Formato de dato programable a 8 ó 9 bits.
- Enabled separado para transmisión y recepción.
- Polaridad de la transmisión de salida programable.
- Interrupt-driven operación con cada bandera.
- Detección de ruido 1/16 bit-time.
- Propósito general, I/O habilidad.

2.5.3 DIAGRAMA DE BLOQUES

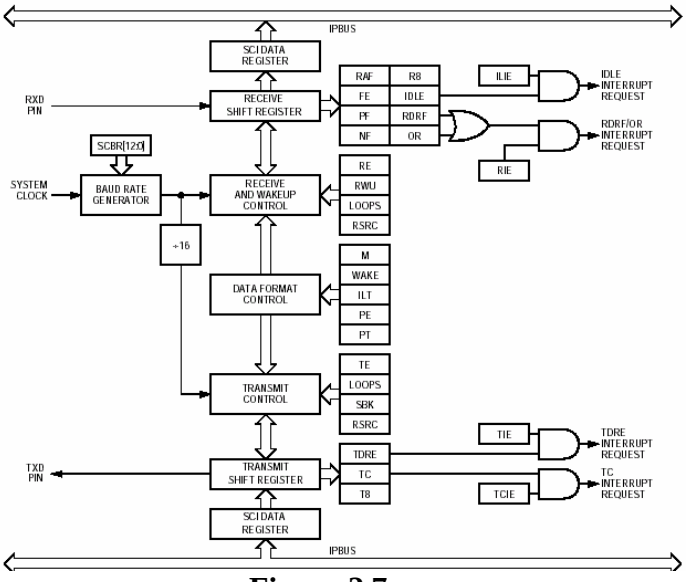

Figura 2.7

2.5.4 DESCRIPCIÓN DE LA SEÑAL

La siguiente tabla da una visión de las señales que se van a describir:

Name	Function	Port	Reset State	Pullup
RXD	Receive data pin	SCIPORT0	0	Disabled
TXD	Transmit data pin	SCIPORT1	0	Disabled

Tabla 2.7

2.5.4.1 RXD

RXD es el pin de recepción del puerto SCI. RXD está disponible para propósito general I/O cuando no es configurado como operación de recepción.

2.5.4.2 TXD

TXD en el pin de recepción del puerto SCI. TXD está disponible para propósito general I/O cuando no es configurado como operación de transmisión.

2.5.5 FORMATO DEL DATO

El SCI usa el estándar NRZ Cada trama tiene un bit de start, ocho o nueve bits de datos y uno o dos bits de stop. Limpiando el bit M del registro SCCR1 se configura el SCI como una trama de diez bits. Si se escribe un uno en este bit el SCI quedará configurado con una trama de 11 bits.

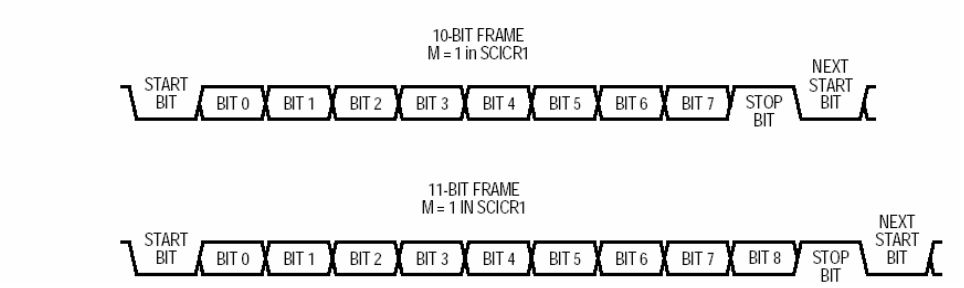

Figura 2.8

2.5.6 BAUD RATE GENERATION

El valor escrito desde 0 a 8191 escrito en SCIBDH y SCIBDL determinan el sistema divisor del reloj.

El BAUD RATE GENERATION está sujeto a dos fuentes de error:

- El divisor del módulo reloj puede no dar exactamente la frecuencia deseada.
- La sincronización con el bus del reloj puede causar desfases.

SBR[12:0]	Receiver Clock (Hz)	Transmitter Clock (Hz)	Target Baud Rate	Percent Error
0x0012	1,833,333.3	114,583.3	115,200	0.54
0x0024	916,666.7	57,291.7	57,600	0.54
0x0036	611,111.1	38,194.4	38,400	0.54
0x003d	540,983.6	33,811.4	33,600	0.63
0x0048	458,333.3	28,645.8	28,800	0.54
0x006b	308,411.2	19,275.7	19,200	0.39
0x0008f	230,769.2	14,423.1	14,400	0.16
0x00d7	153,488.4	95,93.0	9,600	0.07
0x01ae	76,744.2	4,796.5	4,800	0.07
0x035b	38,416.8	2,401.0	2,400	0.04
0x06b7	19,197.2	1,199.8	1,200	0.01
0x0d6d	9,601.4	600.1	600	0.01
0x1adb	4,800.0	300.0	300	0

Tabla 2.8

3 PROTOCOLOS UTILIZADOS

Los protocolos requeridos para este proyecto son IP en la capa de red y TCP en la capa superior a ésta ya que son los que proporcionan una mayor seguridad a la hora de intercambiar paquete, existiendo corrección de errores y retransmisión de paquetes si existe pérdida de éstos. En el nivel físico se utilizará PPP.

A continuación se procederá a explicar detenidamente la estructura de estos protocolos, cómo funcionan…

3.1 PROTOCOLO IP

3.1.1 INTRODUCCIÓN

IP es un protocolo diseñado para interconectar sistemas intercambiando paquetes. Cada datagrama (bloque de datos) se trata de forma independiente respecto al resto de los datagramas.

El protocolo Internet implementa dos funciones básicas que son direccionamiento y fragmentación.

Se usarán las direcciones que se encuentran en la cabecera Internet para transmitir los datagramas de un origen a un destino. Se tendrá que seleccionar un camino para la transmisión de estos paquetes. A ello se le llama encaminamiento.

Puede existir la necesidad de fragmentar un paquete para su transmisión por redes más pequeñas y su posterior reensamblado. Para ello la cabecera dispone de unos campos que informarán de la existencia o no de fragmentación.

Cuatro de los campos de la cabecera que utiliza IP son utilizados para prestar un buen servicio. Son:

- Tipo de servicio: Calidad del servicio deseada.
- Tiempo de vida: Establece el tiempo máximo que un datagrama puede tardar en alcanzar su destino.
- Checksum: Suma de control de verificación. Informa acerca de si los datos transmitidos se han recibido correctamente. Sólo se hace a nivel de cabecera por ello no asegura que la información que transporta esté bien recibida.
- Opciones: Proporcionan funcionalidad de control de algunas situaciones, por ejemplo mecanismos de seguridad, etc.

3.1.2 DESCRIPCIÓN DEL FUNCIONAMIENTO

La función del Protocolo de Internet es encaminar datagramas a través de una serie de redes interconectadas entre sí. Para llevar a cabo este proceso se irán pasando los datagramas desde un módulo de Internet a otro hasta alcanzar el destino.

3.1.2.1 Encaminamiento

Antes de empezar a describir el encaminamiento, hay que saber diferenciar varios conceptos:

- Nombre del host: Nombre del host que se busca.
- Dirección IP: Indicará donde se encuentra el host.
- Ruta: Cómo llegar hasta el host.

El protocolo IP trabaja con direcciones IP. Es tarea de los protocolos de nivel superior traducir nombres de host a direcciones IP.

Una dirección IP está formada por cuatro números enteros, cada uno de ellos tiene un tamaño de un byte y están separados por un punto. Cada dirección IP se compone de dos partes: La primera identifica una red y la segunda a un host concreto dentro de esta red. Dependiendo del número de bits que contenga cada parte se podrá distinguir cuatro redes distintas:

- Redes clase A: Un byte identifica la red y los tres bytes siguientes identifican a los hosts dentro de dicha red. El bit más significativo del primer byte (El que identifica la red) tendrá el valor cero, por ello existen 126 posibles redes y cada una de ellas puede direccionar más de 16 millones de host con los tres bytes de la segunda parte de la dirección.

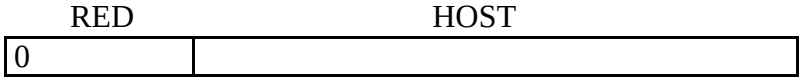

Figura 3.1

- Redes clase B: utiliza dos bytes para identificar la red y los dos bytes restantes para identifican a los hosts dentro de la red. Los dos bits más significativos del primer byte identificador de la red, tendrán el valor uno y cero respectivamente. Con esto, el rango de redes posibles va desde 128.1 hasta 191.254 (191 primer byte y 254 segundo byte). Con los dos bytes que restan se pueden representar más de 65000 host.

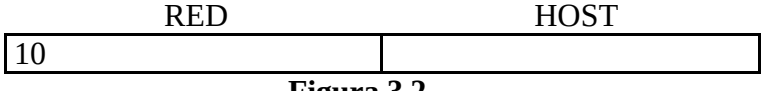

Figura 3.2

- Redes clase C: Estas redes utilizan tres bytes para identificar la red y sólo uno para identificar los hosts. De los bits que identifican la red, los tres más significativos deberán tener los siguientes valores respectivamente uno, uno y cero. Con ello, las redes que se podrán direccionar irán desde 192.1.1 hasta 223.254.254. con el byte que representa a los host se podrán identificar hasta 254 equipos.

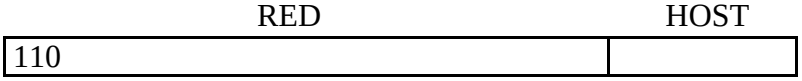

Figura 3.3

- Redes clase D: Son redes con dirección IP especial. Esta dirección está reservada para grupos multienvío. Su primer byte puede valer cualquier número entre 224 y 239. Este valor identificará la red y los tres bytes restantes identifican el número multienvío.

El número de redes que se pueden direccionar en cada clase de red es menor que el que número que se ha comentado. Ello es debido a que existen direcciones reservadas para propósitos especiales.

La siguiente tabla nuestra una serie de direcciones especiales:

Dirección	Descripción
0.0.0.0	Se utiliza con origen de una solicitud de configuración de arranque
127.0.0.0	Reservada
127.0.0.1	Interna (loopback). Cliente y servidor están en la misma máquina
127.0.0.2 – 127.255.255.255	Reservadas
128.0.0.0	Reservada
191.255.0.0	Reservada
192.0.0.0	Reservada
10.0.0.0 – 10.255.255.255	Clase A reservada para Intranet
172.16.0.0 – 172.31.255.255	Clase B reservada para Intranet
192.168.0.0 – 192.168.255.255	Clase C reservada para Intranet
255.255.255.0	Reservada
254.0.0.0 – 255.255.255.254	Reservada
255.255.255.255	Broadcast a todos los nodos de la red

Tabla 3.1

Si se quisiera enviar un mensaje a todos los nodos de la red, se colocarían todos los bits destinados a identificar los host con valor 1. Esta técnica es conocida como broadcast.

3.1.2.2 Máscara de subred

El número que bits que identifica a una red es fijo en las redes de clase A, B o C. Una empresa podría tener la necesidad de variar este tamaño y decir cuantos bits quiere destinar para identificar la red y cuantos para identificar el host.

La máscara de subred es una secuencia de 32 bits, dentro de los cuales aquellos que tengan el valor uno identifican la red y los que tengan el valor cero identificarán los host. Por ejemplo si la máscara de subred es 11111111.11111111.00000000.00000000 identifica a una red de tipo B ya que está diciendo que los 16 primeros bits identifican la red (los dos primeros bytes) y los 16 siguientes identifican al host.

3.1.2.3 Fragmentación

La fragmentación de un datagrama es necesaria cuando el tamaño de éste es mayor que el tamaño máximo del datagrama de la red que tiene que atravesar para llegar al destino.

Un datagrama puede estar marcado para que no se pueda fragmentar. Si es así y llega a un sitio en el que debe ser fragmentado, este datagrama se descartaría. En este proyecto se va a considerar que los datagramas no se puede fragmentar. El tamaño máximo posible viene dado por el tipo de red.

3.1.3 FORMATO DE LA CABECERA

A continuación se observa el formato de la cabecera:

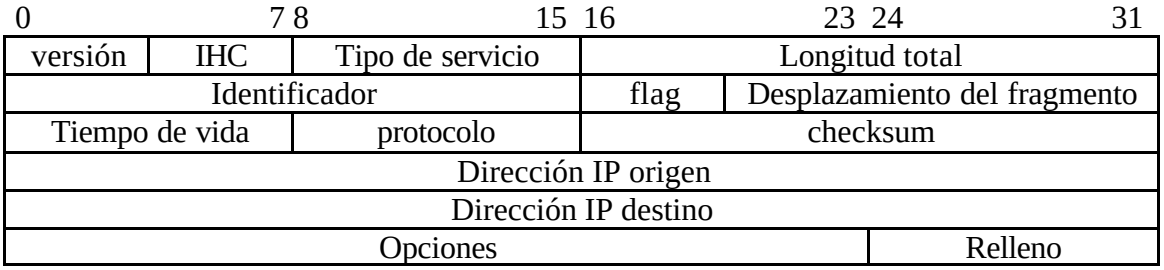

Figura 3.4

- Versión: 4 bits. Describe el formato de la cabecera de Internet. En el proyecto se describe la versión actual, que es la 4. Existen algunas zonas dentro de Internet que están experimentando con la versión 6.
- IHC: 4 bits. Longitud de la cabecera (Internet Header Length). Indica el número de palabras de 32 bits que tiene la cabecera IP. Conociendo este número se sabe cuánto ocupa el campo de opciones. Por defecto, si no existen opciones, la cabecera ocupa 5 palabras.

- Tipo de servicio: 8 bits. Informa de qué hacer con un datagrama en caso de tener que tomar una decisión. Por ejemplo, cuando un router se queda sin memoria para tratar la información que le va llegando, tendrá que descartar datagramas. En este caso, este campo es importante ya que se observará el bit que indica la fiabilidad a 1 y este datagrama tendrá menos posibilidades de ser descartado. La elección más común es un compromiso entre baja demora, alta fiabilidad y alto rendimiento. Los puntos a tener en cuentan serán:
  - Precedencia: medida independiente de la importancia de este datagrama.
  - Retraso: La entrega inmediata es importante para datagramas con esta indicación.
  - Rendimiento: Una alta velocidad de datos es importante para datagramas con esta indicación.
  - Fiabilidad: Para datagramas con esta indicación es importante un mayor esfuerzo para asegurar la entrega.

A continuación se observa como funciona cada uno de los bits:

0	1	2	3	4	5	6	7
Precedencia			D	T	R	0	0

Figura 3.5

El bit D indica retraso normal si vale 0 y retraso bajo si vale 1.
El bit T indica rendimiento normal si vale 0 y rendimiento alto si vale 1.
El bit R indica fiabilidad normal si vale 0 y fiabilidad alta si vale 1.
Los dos últimos bits (el 6 y el 7) están reservados para usos futuros y van colocados a cero.

El campo de precedencias está formado por tres bits:

111	Control de Red
110	Control entre redes
101	Crítico/ECP
100	Muy urgente
011	Urgente
010	Inmediato
001	Prioridad
000	Rutina

Tabla 3.2

En el proyecto que se acomete, el tipo de servicio estará completamente a cero ya que todos los datagramas van a tener igual fiabilidad, rendimiento y retraso. Ello es debido a que estos valores pueden incrementar (en cierto sentido) el coste del servicio. En muchas redes un mejor rendimiento para uno de estos parámetros conlleva un peor rendimiento de algún otro.

- Longitud total: 16 bits. Es el tamaño total del datagrama tanto de la cabecera como los datos, medido en octetos. Permite que la longitud máxima sea 65.535 octetos aunque un datagrama de esta longitud no es práctico. Los hosts deben estar preparados para soportar datagramas de longitud hasta 576 octetos. Se recomienda que no se envíen datagramas de mayor longitud ya que no se sabe si el destinatario está preparado para aceptarlos.

- Identificador: 16 bits. Este campo tiene sentido cuando se ha llevado a cabo la fragmentación IP. Cuando un datagrama está fragmentado, todos los datagramas resultantes de dicha fragmentación tienen el mismo número en esta campo para que un host pueda reconocer que los fragmentos pertenecen al mismo datagrama.

  En este proyecto no se va a considerar la fragmentación, por tanto, este campo será cero.

- Flags: 3 bits. Son indicadores de control.

Figura 3.6

  El más significativo de los bits está reservado, por ello se pone a un valor cero. El segundo es el bit DF (Disable Fragmentation), indica si la fragmentación es posible o no. Cuando vale uno indica que no es posible y si vale cero sí lo es. Cuando un datagrama ha sido fragmentado, todos los fragmentos llevan el tercer bit (MF, More Fragments) a uno excepto el último que lo deberá llevar a cero para indicar que ha terminado el datagrama.

- Desplazamiento del fragmento: 13 bits. Si existe fragmentación, este campo indica qué posición ocupan los bytes respecto al fragmento original.

- Tiempo de vida: 8 bits. TTL: (Time To Live). Indica el tiempo máximo que un datagrama puede estar circulando por la red sin alcanzar su destino. En cada nodo este valor se va decrementando hasta que llega a cero, en ese momento se descartará.

- Protocolo: 8 bits. Este campo indica el protocolo del siguiente nivel usado en el campo de datos del datagrama IP. En la siguiente tabla se observan algunos de los posibles valores según el protocolo.

Número	Tipo	Descripción
1	ICMP	Transmite mensajes de error
2	IGMP	Para grupos de multienvío
6	TCP	Protocolo de control de transmisión
8	EGP	Utilizado para encaminar datagramas a redes externas
17	UDP	Protocolo de Datagramas de usuario
88	IGRP	Protocolo de intercambio de encaminamiento

Tabla 3.3

- Checksum: 16 bits. Suma de control de la cabecera. Permite verificar la integridad de los datos. Debe ser recalculada y verificada en cada punto donde la cabecera IP es procesada.
  - El algoritmo de la suma de control es el siguiente: realizar el complemento a uno de la suma del complemento a uno de todos los campos de la cabecera tomados en palabras de 16 bits. Cuando se calcula la suma, el campo checksum debe estar a cero.
- Dirección IP origen: 32 bits. Dirección del host que envió el datagrama.
- Dirección IP destino: 32 bits. Dirección del host al que va dirigido el datagrama.
- Opciones: Variable. El campo de opciones no tiene por qué aparecer en un datagrama. En algunos entornos la opción relacionada con la seguridad puede ser obligatoria. Si existe este campo la longitud es variable. Si dicha longitud no completa una palabra de 32 bits, se rellenará con ceros hasta completar una palabra.

  Las opciones pueden venir representadas de dos formas:
  - Un único byte que indica el tipo de opción.
  - Un byte que indica el tipo de opción, un byte que indica la longitud de la opción y los bytes relacionados con la opción en sí.

  El byte que indica el tipo de opción estará dividido en tres campos:

Bits	Descripción
1 bits	Flag de copia
2 bits	Opción de clase
5 bits	Opción de numeración

Tabla 3.4

El flag de copia indica si la opción debe ser o no copiada a todos los fragmentos resultantes de la fragmentación (si es que la hubiera). Un cero indica que la opción no debe ser copiada, un uno indica que sí debe ser copiada.

Las opciones de clase son:

- 0 = Control
- 1 = Reservado para usos futuros
- 2 = Depurado y medidas
- 3 = Reservado para usos futuros

Existen diferentes opciones:

Clase	Número	Longitud	Descripción
0	0	-	Final de la lista de opciones
0	1	-	No operación
0	2	11	Seguridad
0	3	Variable	Encaminamiento de origen variable
0	9	Variable	Encaminamiento de origen estricto
0	7	Variable	Registro de rutas
0	8	4	Identificador de stream
2	4	Variable	Marca de tiempo

Tabla 3.5

- Relleno: Variable. El relleno de la cabecera es necesario cuando existen opciones y éstas no son múltiplos de 32bits. Se rellenará con ceros hasta completar una palabra de 32 bits.

Nota: como se ha ido comentando anteriormente, para este proyecto no se considerará fragmentación ni opciones.

3.2 PROTOCOLO TCP

3.2.1 INTRODUCCIÓN

El Protocolo de Control de Transmisión (TCP) es el método más eficiente y seguro de mover tráfico de red entre un cliente y un servidor o entre subredes en general. TCP es un protocolo de propósito general que, además de usarlo sobre IP, como se desarrolla en este proyecto, se puede adaptar para utilizarlo con otros sistemas de entrega.

IP proporciona a TCP un medio para enviar y recibir mensajes envueltos en datagramas de Internet. Éste último proporciona un medio de direccionar paquetes TCP de un origen a un destino situados en redes diferentes.

La interfaz que presenta TCP es, por un lado, con el usuario o los procesos de aplicación, por otro lado TCP se encuentra con un protocolo de nivel más bajo, en este caso será IP. La interfaz con el usuario consiste en una serie de llamadas para manipular ficheros como son abrir o cerrar una conexión, enviar o recibir datos…

3.2.1.1 Áreas que se tratarán

- Transferencia básica de datos: TCP debe ser capaz de transmitir un flujo continuo de octetos entre dos usuarios en ambos sentidos.
- Fiabilidad: TCP debe poder recuperar los datos que se corrompan, se pierdan… Esto se consigue de varias formas:

- Se asigna a cada octeto que se transmita un número de secuencia y quedará a la espera de un acuse de recibo del par remoto. Si no llega un acuse de recibo o ACK transcurrido un determinado tiempo, se supone que los datos enviados se han perdido y se vuelven a retransmitir. El número de secuencia del primer octeto de datos de un segmento se transmite con ese segmento, es lo que se ha llamado número de secuencia. El número de acuse de recibo es el número de secuencia del siguiente octeto de datos que se espera recibir en el par remoto, es decir, será el número de secuencia del siguiente octeto que se debe enviar. Este número indica que los octetos hasta ese número han sido aceptados.
- Para saber si un dato es el correcto o no, cada segmento transmitido lleva asociado un campo de suma de control (checksum). Si el receptor calcula el checksum de los datos que le llegan y comprueba que este suma no es correcta los descarta, serán segmentos dañados.

- Multiplexación: Existen una serie de direcciones o puertos dentro de cada host para permitir que varios procesos dentro de este host puedan establecer cada uno una conexión.
- Conexiones: Los módulos de TCP deben mantener información de control de cada flujo de datos que envía. Se denominará conexión a la combinación de esta información de control junto con las direcciones IP origen y destino, el puerto local y remoto, el número de secuencia y el número que se espera recibir. Toda esta información vendrá almacenada en una estructura que se llamará "bloque de control de transmisión" (TCB, 'Transmission Control Block'). Cuando dos procesos se quieren comunicar primero tiene que crear una conexión e inicializar todos estos parámetros. Cuando la comunicación se termine y se cierre la conexión se debe borrar toda la información almacenada y dejar esa conexión libre para otra posible comunicación.

  Para disponer de direcciones únicas dentro de cada TCP, una conexión vendrá identificada por una pareja formada por una dirección IP y un número de puerto, así se tendrá una dirección de conector ('socket') que será única a lo largo de todo el conjunto de redes interconectadas.
- Prioridad y seguridad: El nivel de prioridad y seguridad en una comunicación pueden ser indicadas por los usuarios de TCP. Si no se necesitan se emplean los valores por defecto.

3.2.1.2 Modo de operación

Cuando se quieren enviar datos, el proceso de envío de TCP los empaqueta en un segmento poniéndole la cabecera TCP. Una vez que se tiene un paquete TCP se pasa al protocolo IP. Éste le colocará la cabecera IP y transmitirá el segmento al TCP destino. Una vez que el receptor recibe el paquete, primero detecta si es para él o no. Comprobará el checksum y si está bien comprobará el número de secuencia para ver si es el paquete esperado. Si todo está bien creará una respuesta enviando en ella datos (si los hubiera).

3.2.2 FORMATO DE LA CABECERA

A los datos que se quiere enviar al host remoto hay que añadirle la cabecera TCP que a continuación se especifica. Este segmento se le pasará a IP. La cabecera IP transporta, entre otras cosas, las direcciones de los host origen y destino que ayudarán a la hora de direccionar un datagrama, como se comentó en el aparatado correspondiente. Una cabecera de TCP sigue a la cabecera de IP, incluyendo información específica de TCP.

Esta cabecera está formada generalmente por 20 bytes de longitud, aunque este campo puede variar si existen opciones.

A continuación se observa el formato de la cabecera:

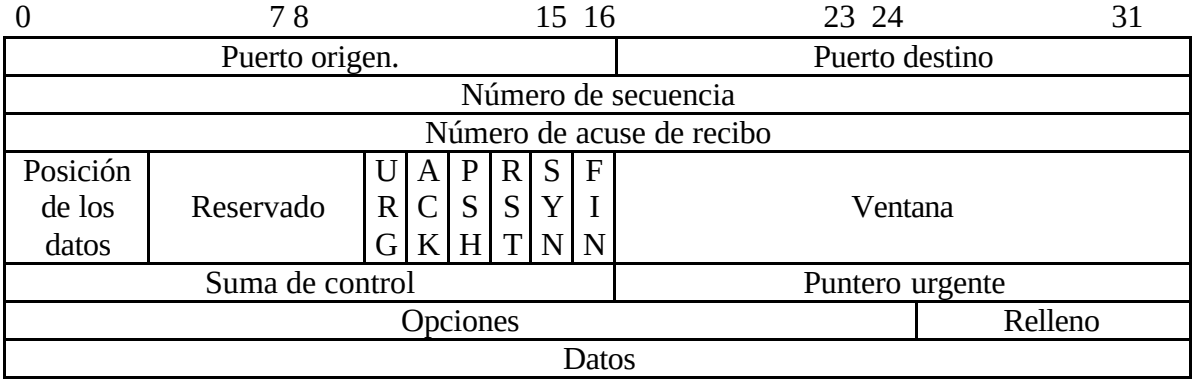

Figura 3.6

- Puerto origen: 16 bits. Número de puerto origen. Cuando un cliente intente realizar una conexión a un puerto de un servidor, se le asignará un número de puerto local.
- Puerto destino: 16 bits. Número de puerto del servidor al cual un cliente se quiere conectar.
- Número de secuencia: 32 bits. Número de secuencia del primer octeto de datos que se transmite. En el establecimiento de la conexión éste será un valor aleatorio y a partir de él se formarán los siguientes números de secuencia.
- Número de acuse de recibo: 32 bits. Valor del número de secuencia que el emisor espera recibir en el siguiente envío. Este campo permite validad los diferentes segmentos que van llegando ya que valida todos los octetos hasta ese número.

La siguiente figura muestra el funcionamiento del número de secuencia y del número de asentimiento. Se supone que el tamaño de los datos que quieren enviar cada host es de 1000 bytes. En el primer paso el host A envía 1000 octetos empezando por el 1023 y espera que el par le envíe el número de octeto 5000 asintiendo así todos los anteriores. En el siguiente paso, el host B envía 1000 octetos siendo el primero de ellos el número 5000 e informa al host A que espera recibir el byte 2023 y que, por tanto, asiente todos los anteriores.

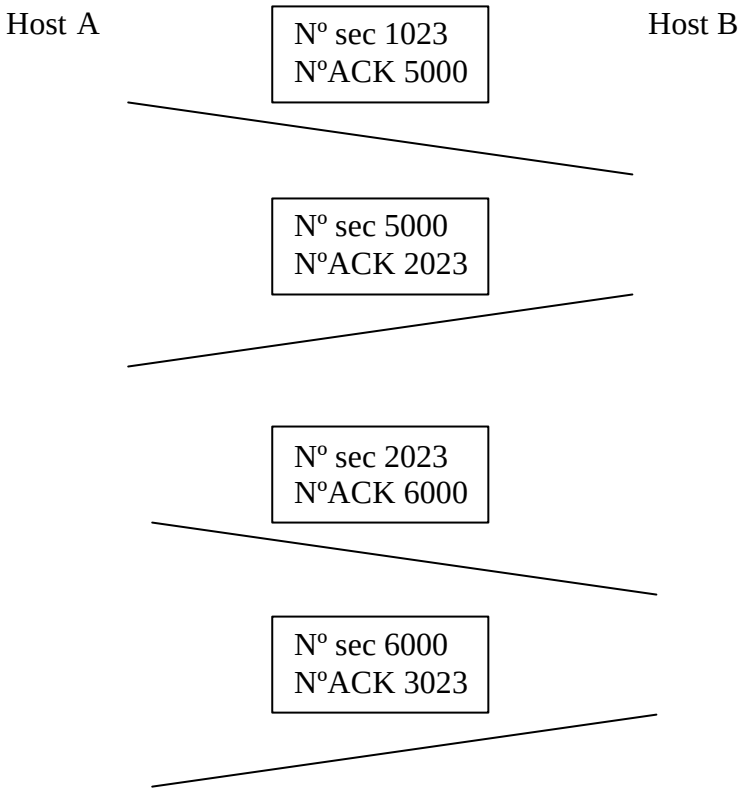

Figura 3.7

- Posición de los datos: 4 bits. Indica el tamaño de la cabecera medida en palabras de 32 bits (La cabecera ocupará siempre un número entero de palabras de 32 bits, aunque lleve opciones). Conociendo este número se sabe dónde comienzan los datos. Si no existen opciones, este campo tendrá un valor de cinco. Si sí existieran, la longitud sería variable. Por ello este campo es obligatorio.
- Reservado: 6 bits. Bits reservados para un uso futuro.
- Bits de control o banderas: 6 bits. Se utiliza para determinar el propósito y contenido del segmento. Se procede a estudiar cada uno de estos bits (de izquierda a derecha):
  - URG: Informa si el segmento transporta datos urgentes. Hace significativo el campo " Puntero urgente".
  - ACK: Indica que el número que se encuentra en el campo 'número de acuse de recibo' o 'número de secuencia ack' de la cabecera es válido. Hace significativo el campo "Número de acuse de recibo".
  - PSH: Indica que los datos transportados deben ser enviados a la aplicación lo antes posible. Función de "entrega de datos inmediata".
  - RST: La conexión debe ser reinicializada ('reset').

- SYN: Proceso de sincronización durante la conexión. Hay que sincronizar los números de secuencia.
- FIN: Se activa cuando el host desea finalizar la conexión. Últimos datos del emisor.

- Tamaño de ventana: 16 bits. Informa del número de bytes que un host está dispuesto a recibir sin tener que validar dicha información. El tamaño puede variar en cualquier momento mientras la conexión esté activa.

- Checksum o suma de control: 16 bits. Contiene una suma de verificación de la cabecera TCP más los datos más una pseudocabecera. La forma de realizar el checksum es la siguiente:

  - Es el complemento a uno de la suma de los complementos a uno de toda la cabecera TCP junto con los datos tomando palabras de 16 bits. Si un segmento contiene un número impar de octetos de cabecera y texto, el último octeto se rellena con ceros a la derecha para formar una palabra de 16 bits y así poder calcular la suma de control. Al calcular dicha suma el campo checksum se considerará formado por ceros.

    A la hora de calcular la suma de control, además de la cabecera y los datos, hay que incluir una pseudocabecera de 96 bits que contiene dirección origen, la dirección destino, el protocolo y la longitud del segmento TCP (longitud de la cabecera TCP más longitud de los datos). El propósito de añadir esta información es el de proporcionar una protección ante segmentos mal encaminados.

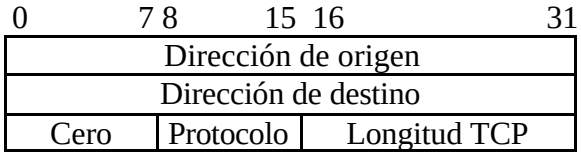

Figura 3.8

- Puntero urgente: 16 bits. Se utiliza en combinación con la bandera URG, es decir, será interpretado únicamente si el bit de control URG está establecido a uno. Apuntará al número de secuencia del octeto al que seguirán los datos urgentes. Se debe notificar la llegada de datos urgentes al programa receptor tan pronto como sea posible, cualquiera que sea su posición dentro del paquete TCP.

- Opciones: Variable. Los campos de opciones pueden ocupar un cierto espacio al final de la cabecera de TCP, pero siempre de una longitud múltiplo de 8 bits.

  En el cálculo de la suma de control, se incluyen todas las opciones. Una opción puede empezar en cualquier posición múltiplo de ocho.
  Existen dos posibilidades para el formato de una opción:

  - Caso 1: Un octeto único con el tipo de opción.

- Caso 2: Un octeto con el tipo de opción, un octeto con la longitud de la opción, y los octetos con los datos propiamente dichos de la opción.

La longitud de la opción tiene en cuenta tanto el octeto con el tipo de opción como el propio octeto de longitud así como los octetos con los datos de la opción.

- Relleno: Variable. El relleno de la cabecera es necesario cuando existen opciones y éstas no son múltiplos de 32bits, es decir, que los datos empiezan en una posición múltiplo de 32 bits. Se rellenará con ceros.

3.2.3 DESCRIPCIÓN DEL FUNCIONAMIENTO

3.2.3.1 Estados posibles

Para empezar a describir el funcionamiento del protocolo TCP hay que saber que una conexión avanza de acuerdo a una serie de estados durante su tiempo de vida. Los estados son los siguientes; 'LISTEN' (en escucha), 'SYN_SENT' (SYN enviado), 'SYN_RECEIVED' (SYN recibido), 'ESTABLIHESD' (establecida), 'FIN_WAIT_1' (en espera de fin-1), 'FIN_WAIT_2' (en espera de fin-2), 'CLOSE_WAIT' (en espera de cierre), 'CLOSING' (cerrándose), 'LAST –ACK' (TIME_WAIT' (en espera) y 'CLOSED' (cerrada). Este último estado es ficticio ya que representa un estado en el que no existe bloque de control de transmisión, es decir, no existe una estructura que mantenga la información de una conexión ya que no ésta existe. A continuación se desarrollan, de forma breve, los significados de los estados aunque se entenderán mejor al desarrollar el modo de funcionamiento:

- LISTEN: Estado en el que se está a la espera de una solicitud de conexión.
- SYN_SENT: A este estado se llega tras enviar una solicitud de conexión y estar a la espera de otra solicitud por parte del host remoto.
- SYN_RECEIVED: Representa la espera del acuse de recibo confirmando la solicitud de conexión tras haber, tanto recibido como enviado una solicitud de conexión.
- ESTABLIHED: Estado en el que la conexión está abierta, los datos recibidos pueden ser entregados al usuario y éste puede enviar datos al par remoto. Es el estado normal para la transferencia de información en una conexión.
- FIN_WAIT_1: Estado en el que la conexión está a la espera de una solicitud de finalización de la conexión proveniente del TCP remoto, o también puede estar esperando el acuse de recibo de la solicitud de finalización previamente enviada.
- FIN_WAIT_2: Representa la espera de una solicitud de finalización del TCP remoto.

- CLOSE_WAIT: La conexión está a la espera de una solicitud de finalización de la conexión proveniente del usuario local.
- CLOSING: Estado en el que se está a la espera del paquete, proveniente de TCP remoto, con el acuse de recibo de la solicitud de finalización.
- LAST_ACK: Representa la espera del acuse de recibo de la solicitud de finalización de la conexión tras haber enviado ésta al TCP remoto (lo que incluye haber enviado el acuse de recibo de la solicitud remota de finalización de la conexión.
- TIME_WAIT: La conexión espera durante suficiente tiempo antes de cerrarse para asegurar que el TCP remoto recibió el acuse de recibo de su solicitud de finalización de la conexión.
- CLOSED: Estado en el que no existe conexión, se ha borrado el bloque de control de transmisión.

Una conexión pasa de un estado a otro en respuesta a eventos que serán las llamadas de usuario:

- mti_listen ( ): poner un puerto a la escucha.
- mti_connect ( ): abrir una conexión.
- mti_accept ( ): aceptar una solicitud de conexión.
- mti_read ( ): recibir en paquete.
- mti_write ( ): almacenar datos que se quieren enviar
- mti_close ( ): cerrar una conexión.
- mti_abort ( ): interrumpir una conexión.

3.2.3.2 Números de secuencia

Una de las nociones fundamentales del diseño de TCP es el hecho de que cada octeto de datos enviado por una conexión tenga asociado un número de secuencia y por ello podrá tener un acuse de recibo. El mecanismo que se emplea es acumulativo, es decir el acuse de recibo de un número determinado asiente todos los anteriores, es decir, informa de que han sido recibidos, así no se tiene que asentir octeto a octeto. Este mecanismo permite la detección fácil y directa de duplicados generados por retransmisiones. La numeración de los octetos es de la siguiente forma: El primer octeto de datos tras la cabecera es el de menor número y los siguientes octetos se van numerando consecutivamente.

3.2.3.3 Diagrama de estados

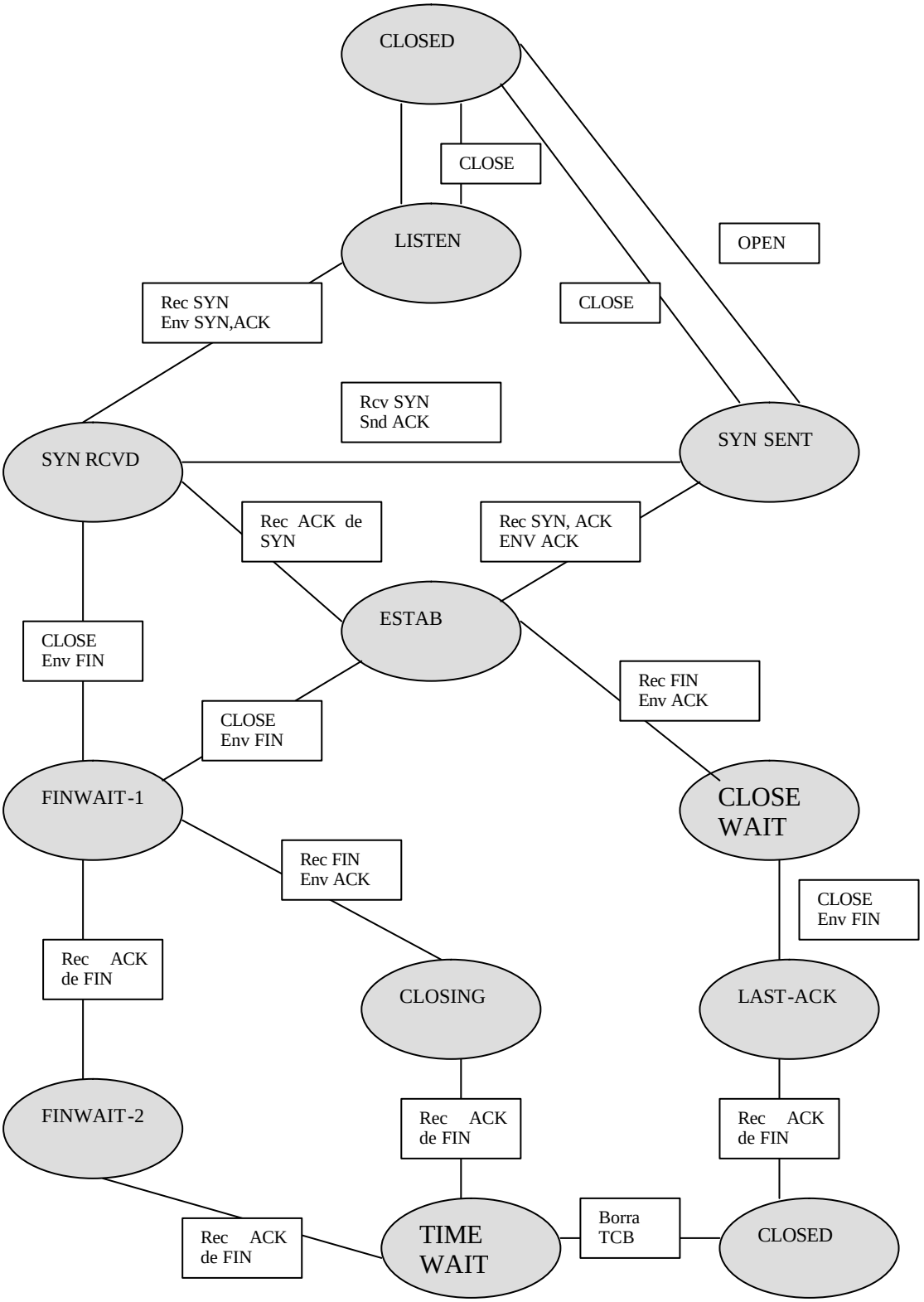

Figura 3.9

3.2.3.4 Establecimiento de la conexión TCP

Una conexión TCP requiere un procedimiento en tres fases bien diferenciadas:

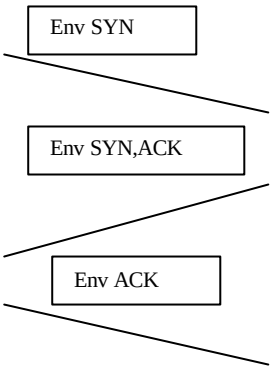

Figura 3.10

Normalmente, este proceso lo inicia un TCP cliente y responde un TCP servidor. En estas tres fases se produce la sincronización de la conexión, que requiere que cada parte envíe su propio número de secuencia inicial y que reciba una confirmación de su llegada en la forma de acuse de recibo de la otra parte. Cada parte debe también recibir el número de secuencia inicial de la otra parte y enviar un acuse de recibo como confirmación.

Un servidor debe estar preparado para recibir una petición de conexión, para ello deber haber llamado a las funciones mti_listen( ) para que el servidor esté preparado para recibir la petición.

Cuando un cliente genera una llamada mti_connect( ) envía un paquete al servidor solicitando abrir una conexión. Este paquete lleva activo el bit SYN de las banderas para indicar al servidor que está en proceso de sincronización. Dicho segmento no llevará ningún dato, sino únicamente la cabecera IP, la cabecera TCP y las posibles opciones. En el campo número de secuencia llevará un valor que será el número de secuencia inicial.

¿Qué problema sucede en este momento? Si la conexión se abre y se cierra en una sucesión muy rápida o si la conexión se corta acompañada de una pérdida de la información ya enviada y después se reestablece pueden aparecer segmentos duplicados. Para evitar una confusión, se debe intentar que los números de secuencia inicial sean distintos en cada conexión, evitando así que una conexión utilice un número que ya está siendo utilizado en la red. Para solucionar esto se utilizará un generador de números de secuencia iniciales.

El servidor responderá enviando un segmento de aceptación al segmento anterior. Ello se realiza activando la bandera ACK y en el campo número de acuse de recibo se coloca el valor correspondiente al campo número de secuencia del segmento recibido pero incrementado en una unidad. La bandera SYN también viaja activada para indicar que el proceso de sincronización no ha finalizado todavía. En el campo número de secuencia irá un número de secuencia inicial que será distinto al que se generó anteriormente y también será aleatorio. Cuando el cliente recibe este paquete es el momento en el que éste sabe que

el servidor a validado su petición, pero el servidor está esperando que se valide su segmento.

El cliente enviará un paquete que validará el enviado por el servidor. En este paquete viajará la bandera ACK activada colocando el valor correspondiente en el campo número de acuse de recibo. La bandera SYN no viajará activa en esta ocasión.

TCP A		TCP B
1. CLOSED		LISTEN
2. SYN-SENT	→ SEQ = 100, CTL = SYN →	SYN-RECEIVED
3. ESTABLIHSED	← SEQ=300, ACK=101,CTL=SYN,ACK ←	SYN-RECEIVED
4. ESTABLIHSED	→ SEQ=101, ACK=301,CTL=ACK →	ESTABLIHSED

Es en este momento cuando se tiene la conexión establecida correctamente.

3.2.3.5 Flujo de datos

Cuando ya se ha establecido la conexión los datos se transmiten mediante el intercambio de segmentos. Para que el protocolo TCP sea fiable se utilizan los números de secuencia y números de acuses de recibo. Como los segmentos pueden perderse debido a errores o congestión en la red, TCP puede retransmitir los segmentos (tras un tiempo de espera) para asegurar su entrega. También pueden existir segmentos duplicados debido a la red o a la retransmisión. TCP realizará comprobaciones para verificar que los segmentos son admisibles.

El emisor de los datos en su estructura o bloque de control mantiene información necesaria para la comunicación como es el próximo número de secuencia a enviar, el número de secuencia menor sin acuse de recibo, número de puerto, etc.

Cuando llega un paquete primero se comprueba si el paquete es para la conexión activa comprobando las direcciones IP y el número de puerto. Después se comprueba si el número de acuse de recibo es el que se espera, el que está almacenado en la estructura. Si todo es correcto se puede enviar otro paquete y para ello se actualizará tanto el número de secuencia que se envía como el que se almacena así como el número de acuse que se envía y el que se almacena.

3.2.3.6 Finalización ordenada de una conexión TCP

Cuando se quiere cerrar una conexión significa que no tiene más datos que enviar. En ese momento el usuario no podrá enviar más datos y la respuesta de cualquier llamada a mti_write( ) dará error. El cliente enviará todos los datos que estén pendientes de enviar y validará todos los datos que le lleguen del servidor hasta que éste envíe el paquete con la bandera FIN.

Cuando un host desea finalizar una conexión de forma ordenada se genera una llamada a mti_close( ). Se generará el envío de un segmento con el bit FIN de las banderas de la cabecera TCP activado, dando a entender la intención de finalizar la conexión.

El servidor se encarga en ese momento de validar cualquier información previa mediante el envío de un segmento con el bit ACK activado. En este punto, el cliente está preparado para cerrar la conexión, pero esto no es posible, ya que tiene que esperar a que el servidor envíe un segmento con la bandera FIN activada. Tras el envío el cliente lo validará. Mientras el servidor no cierre la conexión el cliente puede validar la información que le vaya llegando.

3.2.3.7 Órdenes de usuario y procesamiento de eventos

A continuación se describen las órdenes de usuario al módulo de TCP. El lector debe saber que, debido a que cada sistema operativo tendrá distintos recursos, existen diferentes implementaciones de TCP que pueden tener diferentes interfaces. Todos los TCP deben proporcional un conjunto mínimo de servicios. Se va a describir las funciones más importantes. Éstas especifican funciones básicas que debe ejecutar TCP para soportar la comunicación entre procesos.

Junto con la descripción de las órdenes de usuario se va a estudiar cómo TCP responde a una de estas órdenes según en el estado en el que se encuentre.

- mti_connect( ): El módulo TCP local quiere conectarse a un puerto remoto. Se le pasa como parámetros la dirección IP y el número de puerto del host remoto al que se quiere conectar. Se crea un bloque de control de transmisión que se llenará parcialmente con los datos que se le han pasado. Como no se van a considerar valores de prioridad o seguridad se toman los valores por defecto. TCP devolverá al usuario el nombre de la conexión local.

  Cuando se hace una llamada a mti_connect( ) hay que estudiar en que estado se encuentra la conexión:

  - Estado ‘CLOSED’: Se crea un nuevo bloque de control de transmisión que tendrá la información sobre el estado de la conexión. Se envía un segmento SYN, se selecciona un número de secuencia inicial. Se pasa al estado SYN_SENT y se retorna. Si no hay espacio para crear una nueva conexión se devuelve error.

  - Estados ‘SYN_SENT’, ‘SYN_RECEIVED’, ‘ESTABLISHED’, ‘FIN_WAIT_1’, ‘FIN_WAIT_2, ‘CLOSE_WAIT’, ‘CLOSING’, ‘LAST_ACK’ y ‘TIME_WAIT’ se devuelve error porque la conexión ya existe.

- mti_listen( ): Pone un puerto, que se le pasa como parámetro, a la escucha.

  - Estado ‘CLOSED’: Se crea un nuevo bloque de control de transmisión que tendrá la información sobre el estado de la conexión. Se pasa al estado LISTEN.

- Estados 'SYN_SENT', 'SYN_RECEIVED', 'ESTABLISHED', 'FIN_WAIT_1', 'FIN_WAIT_2, 'CLOSE_WAIT', 'CLOSING', 'LAST_ACK' y 'TIME_WAIT' se devuelve error porque la conexión ya existe.

- mti_accept( ): Acepta una conexión cuando llega un paquete SYN. Se rellena el boque de control de transmisión con la información necesaria que se le ha pasado como parámetro.
  - Estado 'LISTEN': Se crea un nuevo bloque de control de transmisión que tendrá la información sobre el estado de la conexión. Se recibe un paquete con la bandera SYN y se responde validando la conexión con un segmento que contenga activos los bits SYN y ACK. Se pasa al estado SYN_RECEIVED.
  - Estados 'CLOSED' 'SYN_SENT', 'SYN_RECEIVED', 'ESTABLISHED', 'FIN_WAIT_1', 'FIN_WAIT_2, 'CLOSE_WAIT', 'CLOSING', 'LAST_ACK' y 'TIME_WAIT' se devuelve error porque la conexión ya existe.
- mti_read( ): Cuando Se hace esta llamada se lee un paquete que se encontrará en el buffer de entrada. Se procesará el paquete y si es correcto se rellenara el buffer que se le pasa como parámetro con los datos que transportaba dicho paquete. Si cuando se hace esta llamada aún no se ha realizado la llamada a mti_connect( ) se devolverá error porque aún no se tiene una conexión de la que leer datos.
  - Estado 'CLOSED': Se devuelve error ya que el usuario no tiene acceso a esta conexión.
  - Estados 'LISTEN', 'SYN_SENT' y 'SYN_RECEIVED': Se procesa el segmento de entrada y si es correcto se contesta con la correspondiente validación. Según el estado se pasará a 'SYN_RECEIVED' si se encontrara en LISTEN o a ESTABLISHED si estuviera en alguno de los otros dos casos.
  - Estados 'ESTABLISHED', 'FIN_WAIT_1' y 'FIN_WAIT_2: Se procesa el segmento de entrada, si es correcto se extraen los datos que transporten y se contesta con la correspondiente validación incluyendo los datos, si es que los hubiera, que se quieran enviar al otro extremo de la conexión.
  - Estado 'CLOSE_WAIT': El extremo remoto ya ha enviado un FIN, se procesarán los paquetes entrantes y se enviarían segmentos con datos que estuvieran en la cola y que aún no hayan sido enviados.
  - Estados 'CLOSING', 'LAST_ACK' y 'TIME_WAIT' se devuelve error porque la conexión ya se está cerrando.

- mti_write( ): Esta llamada hace que se escriban en una lista de datos a enviar a la conexión, que se le pasa como parámetro, los datos que también se le pasan como parámetros. Si no existiera espacio en la lista para almacenarlos o la conexión aún no esté abierta devolvería error. Los estados posibles en los que

se puede encontrar cuando se realice esta llamada son los siguientes (con esta llamada no se cambia de estado):

- Estado 'CLOSED': Si el usuario no tuviera acceso a esta conexión se devolvería error.
- Estados 'LISTEN': Se devuelve error porque, aunque el puerto esté a la escucha, aún no existe la conexión.
- Estados 'SYN_SENT' y 'SYN_RECEIVED': Se almacena en la cola de datos a enviar y éstos se enviarán una vez que se haya pasado al estado ESTABLISHED.
- Estados 'ESTABLISHED' y 'CLOSE_WAIT': Los datos que se le pasan como parámetros se almacenan en la lista para que se envíen al host remoto. Si no hubiera espacio en la lista para almacenar estos datos se devolvería error.
- 'FIN_WAIT_1', 'FIN_WAIT_2, 'CLOSING', 'LAST_ACK' y 'TIME_WAIT': Se devuelve error porque la conexión se está cerrando y, por tanto no se puede enviar datos a una conexión que no existe.

- mti_close( ): Cuando se hace esta llamada se cierra la conexión especificada. Si la conexión no estuviera abierta, se devuelve error. Si existen datos pendientes de enviar, se transmitirán al par antes de cerrar la conexión. Por lo tanto se pueden hacer llamadas mti_read( ) después de una llamada mti_close( ) para poder enviar los datos pendientes y recibir sus correspondientes validaciones. Una llamada a mti_close( ) significa que "no tengo más datos que enviar" pero no "no recibiré más". Puede suceder que el lado de la conexión que cierra no sea capaz de deshacerse de todos sus datos pendientes de enviar. En este caso, la llamada mti_close( ) se convierte en una llamada mti_abort( ).
  - Estado 'CLOSED': Devolverá error porque no se puede cerrar una conexión que ya está cerrada.
  - Estado 'SYN_SENT': Se borra el bloque de control de transmisión y se devolverá error cerrando a cualquier llamada a mti_write( ) o mti_read( ).
  - Estado 'SYN_RECEIVED': Si no hubiera datos pendientes de ser transmitidos al par remoto se construiría un segmento de FIN y se envía, tras ello se parará al estado FIN_WAIT_1. Si hubiera datos para enviar, no se enviaría el paquete con la bandera FIN. Se pasaría al estado ESTABLISHED y cuando se termine de enviar todos los datos se enviaría el paquete FIN.
  - Estado 'ESTABLISHED': No se enviará el paquete con el FIN hasta que no se termine de transmitir todos los datos pendientes. Una vez que no queden datos, se construye un segmento FIN y se envía. Se pasará al estado FIN_WAIT_1.
  - Estados 'FIN_WAIT_1' y 'FIN_WAIT_2: Este caso es un error, ya que si se encuentra en alguno de estos estados es porque ya se ha hecho una

llamada a mti_close( ). Por ello, si se vuelve a intentar cerrar la conexión, devolverá error.

- Estado 'CLOSE_WAIT': En primer lugar se enviarán todos los datos pendientes y, una vez que no quede ninguno pendiente, se enviará el segmento FIN y se pasa al estado CLOSING.
- Estados 'CLOSING', 'LAST_ACK' y 'TIME_WAIT': Se devuelve error porque no se puede cerrar una conexión que ya se está cerrando.

- mti_abort( ): Esta orden aborta la ejecución de todas las órdenes mti_write() y mti_read pendientes, elimina el bloque de control correspondiente y envía un paquete de RESET al otro lado de la conexión.
  - Estado 'CLOSED': Devolverá error porque no se puede abortar una conexión que no existe.
  - Estado 'LISTEN': Se borra el bloque de control de transmisión. A cualquier orden mti_read( ) se devolverá error ya que se ha abortado la conexión enviado un reset.
  - Estado 'SYN_SENT': Todas las órdenes mti_read( ) y mti_write( ) darían error porque se ha abortado la conexión. Se borra el bloque de control de transmisión, se pasa la estado CLOSED y se retorna.
  - Estado 'SYN_RECEIVED', 'ESTABLISHED', FIN_WAIT_1, 'FIN_WAIT_2 y 'CLOSE_WAIT': Se envía un paquete de RESET. Todas las órdenes mti_read( ) y mti_write( ) devolverán error, se ha abortado la conexión. Al igual que antes, se borra el bloque de control de transmisión y se retorna.
  - Estados 'CLOSING', 'LAST_ACK' y 'TIME_WAIT': Se borrar el bloque de control de transmisión correspondiente a la conexión que se aborta, se pasa al estado CLOSED y se retorna.

3.3 PROTOCOLO PPP

3.3.1 INTRODUCCIÓN

PPP es un protocolo punto a punto cuya función es intercambiar datagramas entre dos host de un enlace de comunicación full-duplex. PPP se ha establecido como el protocolo estándar para acceso a redes TCP/IP a través de línea serie.

Existen tres componentes principales en el protocolo PPP:

- Encapsulado: Se ofrece la posibilidad de multiplexar diferentes protocolos de nivel de red sobre un mismo enlace serie

- Protocolo de control de enlace (LCP): Configurará las opciones de encapsulado, el tamaño de los paquetes, autentificará al otro extremos del enlace, etc.
- Protocolos de control de red (NCPs): Manejarán las particularidades de los diferentes protocolos a nivel de red con los que PPP puede trabajar.

3.3.2 FORMATO DE LA TRAMA PPP

Un paquete PPP tiene la siguiente estructura:

Protocolo	Información	Relleno

Figura 3.11

- Protocolo: 8 ó 16 bits. Su tamaño puede ser uno o dos octetos. Identifica el paquete encapsulado en el campo de información. Hay algunos valores reservados.

  Algunos de los valores que puede tomar este campo se observan en la siguiente tabla:

Código	Protocolo
0x0001	Reservado
0x0021	Protocolo de Internet
0xc021	Protocolo de control de enlace
0xc023	Protocolo de autentificación de password

Tabla 3.5

- Información: Variable. Este campo puede ser desde cero hasta varios bytes. Contendrá el datagrama para el protocolo indicado en el primer campo. El tamaño máximo el campo información viene fijado por el valor de MRU, que tiene un valor por defecto de 1500 bytes, aunque puede tomar otros valores tras el proceso de negociación al establecer la conexión.
- Relleno: Opcionalmente el campo de información podría ser completado con bytes de relleno hasta alcanzar el número de bytes indicado en el MRU.

El paquete PPP debe ser encapsulado a su vez en una trama a nivel de enlace. El formato elegido es el dic, que tiene el siguiente formato:

Flag	Dirección	Control	Datos	Checksum	Flag

Figura 3.12

- Flag: 1 byte. Tiene un valor de 0x7E y marcará el principio y final de la trama.

- Dirección: 1 byte. Este campo tendrá un valor de 0xFF.
- Control: 1 byte. Tendrá un calor de 0x03.
- Datos: Variable. Será el paquete comentado anteriormente.
- Checksum: 2 bytes. Suma de control de verificación. Comprobará si existen errores.

Dentro del campo de Datos, cualquier carácter con valor 0x7E deberá tratarse de forma especial para que no se confunda con los campos de Flag. Para ello, los valores 0x7E que aparezcan en los datos se convertirá en la pareja e valores 0x7D 0x5E. El valor 0x7D identifica el carácter de escape usado para anular los calores con significado especial. Por su parte, el valor 0x5E es el valor original 0x7E pero con su sexto bit cambiado.

El carácter de escape también deberá ser anulado si se encuentra en el campo de datos. El procedimiento es el mismo, al encontrar un valor 0x7D se cambiará su sexto bit, pasando a tener un valor de 0x5D. A es nuevo valor también se le antepondrá el carácter de escape 0x7D.

Cualquier carácter que pueda ser tratado de forma especial y que se encuentre en el campo de datos, deberá ser anulado siguiendo el mismo procedimiento.

3.3.3 ESTABLECIMIENTO DEL ENLACE

Para establecer un enlace, lo primero que tiene que hacer cada host es enviar paquetes LCP que configuren y comprueben el enlace de datos. Cuando ya se encuentra establecido el enlace se procederá a la autentificación si fuera necesario.

Tras esto, es siguiente paso es el envío de paquetes NCP para seleccionar los protocolos de nivel de red que serán encapsulados. Una vez que se ha realizado esto, ya pueden proceder ambos extremos al envío de datagramas.

El enlace se mantendrá hasta que se cierre explícitamente mediante un paquete LCP o NCP o algún suceso externo lo fuerce.

3.3.4 VENTAJAS DE PPP

- Efectúa detección de errores.
- Reconoce múltiples protocolos.
- Se negocian las direcciones IP al momento de la conexión.
- Proporciona verificación de autenticidad.

4 FUNCIONES NECESARIAS PARA IMPLEMENTAR LA PILA TCP/IP

La librería que se va a crear consta de una serie de funciones necesarias para implementar una pila TCP/IP que se dividirán en varios grupos. Las principales serán aquellas que podrá usar el usuario para realizar los distintos procesos, como por ejemplo la lectura de tramas o la escritura de datos para el host remoto. Otra parte de las funciones pertenecerán a un nivel inferior y su función será la de realizar todo el proceso TCP/IP o procesos auxiliares a éste.

Cada función tiene un prefijo característico del proyecto cuyo significado es el siguiente:

- MTI:
  - M: MCORE.
  - T: TCP.
  - I: IP.

Se dividirán de la siguiente manera:

- Funciones de primer nivel: Son las funciones que conoce el usuario. Se utilizaran para realizar tareas como inicializar el sistema, leer tramas, escribir datos… Son las siguientes:
  - mcore_init_system( ): inicializa los registros del microcontrolador y configura las interrupciones.
  - mti_init( ): Es la encargada de inicializar el sistema.
  - mti_listen( ): El objetivo de esta función es poner a la escucha un puerto.
  - mti_connect( ): Se llama cuando un host se quiere conectar a una dirección remota.
  - mti_accept( ): Se acepta una conexión cuando llega un paquete de SYN.
  - mti_read( ): Su objetivo es leer un paquete de entrada, tratarlo y enviar un paquete de respuesta incluyendo datos pendientes(si los hubiera).
  - mti_write( ): Es llamada cuando se quiere mandar información a un puerto remoto.
  - mti_close( ): Cierra una conexión correctamente, esperando a que se terminen de enviar todos los datos que queden pendientes.
  - mti_abort( ): Aborta la conexión que se le pasa como parámetro.

- Funciones de segundo nivel: Son las llamadas por alguna función de primer nivel para seguir con el proceso que estaban realizando:

- mti_encontrada_conexión( ): Es llamada por mti_connect( ). Es la encargada de, una vez encontrado una conexión libre, rellenar parte de los campos de la cabecera.
- mti_trama_completa_rx( ): Es llamada por mti_read( ) y mti_accept( ). Espera hasta que ha llegado un paquete completo para poder procesarlo.
- mti_proceso_estados( ): Es llamada por mti_read( ). Su función es, una vez que ha llegado un dato completo, estudiar el estado en el que se encuentra la conexión y según éste, actuar.
- mti_fin_procesoestados( ): Es llamada mti_proceso_estados( ). Según en el estado al que haya evolucionado la conexión, mandará un paquete de ACK, FIN, datos…
- mti_reset( ): Cuando se aborta una conexión habrá que resetear ésta. Para ello se utiliza esta función.

- Funciones auxiliares. Son funciones utilizadas por todas las funciones anteriores para realizar procesos puntuales. Se dividirán en varios grupos:

  - Funciones relacionadas con el envío o la recepción de una trama:
    - mti_enviar_trama( ): Completa parte de los campos de la trama que se tiene que enviar. Algunos de estos campos que actualiza son el campo de asentimiento, de número de secuencia, puerto origen, puerto destino, dirección IP origen y destino. Llama a la siguiente función para terminar de completar los campos.
    - mti_enviar_cabaecera( ): Completa el resto de los campos. Llamará a las funciones que calculan el checksum tanto de IP como de TCP para poner este valor en su campo correspondiente.
    - mti_convierte_formatoSCI( ): Convierte El buffer de salida, que es una estructura, a una cadena de octetos para poder pasárselo al puerto serie.
    - mti_enviar_ack( ): Envía una trama ACK, sin datos. En el campo banderas se activa el bit correspondiente al ACK.
    - mti_enviar_fin( ): Envía un paquete de FIN. Activa el bit de las banderas correspondiente a FIN
    - mti_enviar_finack( ): Enviará tanto la bandera de ACK como la de FIN.
    - mti_guarda_trama( ): Cuando llega una trama para un puerto distinto al puerto local de la conexión de trabajo, habrá que almacenarla para que otra conexión pueda utilizarla.

- mti_trama_anter( ): Almacena la última trama enviada por si hay que volver a retransmitirla por algún error.

o Funciones que trabajan con listas enlazadas: Se encargan de buscar alguna estructura o algún valor dentro de una estructura, ya sea de la lista de conexiones como de la lista de datos a enviar.

- mti_buscar_conex_libre( ): Busca una conexión libre para poder utilizarla.

- mti_buscar_tcpestadobanderas( ): Busca dentro de la lista de conexiones la primera de ellas que contenga en el campo tcpestadobanderas un valor igual al que se le pasa como parámetro.

- mti_buscar_id_conex( ): Busca dentro de la lista de conexiones aquella que tenga el número identificador de conexión igual al que se le pasa como parámetro.

- mti_buscar_puerto( ): Busca una conexión que tenga en el puerto local el mismo número que el puerto que se le pasa como parámetro.

- mti_borrar_estructura( ): Busca la conexión que tenga por identificador el número que se le pasa como parámetro y la borra.

- mti_buscar_datoenviar( ): Busca dentro de la lista de datos pendientes de enviar la primera estructura que tenga el mismo identificador que el que se le pasa como parámetro. De esta estructura se obtienen bs datos que se quieren enviar.

- mti_borrar_datoenviado( ): Busca la primera estructura, dentro de la lista de datos a enviar, que tenga por identificador el mismo que el que se le pasó como parámetro y borrará dicha estructura.

o Otras funciones:

- mti_checksum( ): Calcula el checksum de una cadena de elementos de tamaño INT8U.

- mti_ipchksum( ): Calcula el checksum de la cabecera IP.

- mti_tcpchksum( ): Calcula el checksum de la cabecera TCP, una pseudocabecera y de los datos que transporte.

- mti_actualizar_rcv_nxt( ): Actualiza el campo acuse de recibo sumándole el número que se le pasa como parámetro.

- mti_actualizar_ack_nxt( ): Actualiza el valor del número de secuencia que se espera recibir.

- mti_malloc_conex( ): Busca una estructura que no esté siendo utilizada para poder almacenar los datos de una nueva conexión.

- mti_free_conex( ): Cuando se cierra una conexión hay que liberar la estructura que se estaba utilizando para que otra conexión la pueda utilizar.

- mti_malloc_enviar( ): Busca una estructura dentro de la lista de datos a enviar, que se pueda utilizar para poder almacenar los datos que el usuario quiere enviar al host remoto.

- mti_free_enviar( ): Se llamará cuando ya no sea necesario enviar al host remoto un dato almacenado. Ello puede ser debido a que ya se ha enviado o se ha abortado la conexión y ya no es necesario su envío. Borrará la estructura que almacene dicho dato.

- Funciones para configurar y utilizar los periféricos del microcontrolador:

  Estas funciones ya han sido diseñadas. Se comenta su funcionamiento para un mejor entendimiento del proyecto.

  - SystemInit: Inicializa todos los registros del MMC2107.
  - PIT1: Función que configura el temporizador 1 para que salte interrupción cuando llegue al final de la cuenta.
  - SCI1: Configura el puerto serie asíncrono para que transmita o reciba datos a una determinada velocidad.
  - enviar_SCI: Envía datos por el puerto serie asíncrono.
  - isr_SCI1_RDRF: Cuando hay una interrupción porque han llegado datos por el puerto serie asíncrono, se salta a esta función y el dato de llagada lo almacena el un buffer.
  - Isr_PIT1_PIF: Cuando el temporizador salta, llama a esta función. Ella incrementará unos contadores necesarios para que el proceso no se quede parado mucho tiempo esperando una respuesta.

4.1 FICHEROS EXISTENTES

Se pasa a describir como se estructurarán todas las funciones anteriores en los ficheros existentes. Se intentará que las funciones que tienen características parecidas se encuentren en un mismo fichero para poder acceder a ellas más fácilmente. No se describirán las funciones ya que se hará detenidamente en puntos sucesivos.

- mti_init.c: Almacena las funciones que se utilizan en la apertura de la conexión. Definirá todas las variables globales y las funciones relacionadas con el microcontrolador.

  - mti_init( ).
  - mti_connect( ).

- mti_encontrada_conexión( ).
- mti_listen( ).
- mti_accept( ).

- mti_mcore_inter.c: Almacena las funciones asociadas a las interrupciones producidas por el microcontrolador, así como la función que lo inicializa.

  - mcore_init_system( ).
  - isr_PIT1_PIF( ).
  - isr_SCI1_RDRF( ).

- Mti_read.c: Contendrá la función necesaria para leer una trama así como las que procesan los estados y la evolución según la información que llegue.

  - mti_read( ).
  - mti_proceso_estados( ).
  - mti_fin_procesoestados( ).

- Mti_write.c: Almacena la función que se necesita para escribir los datos que se quieren enviar al host remoto. También aparecerán en este fichero las funciones de búsqueda y borrado de las estructuras que contienen dichos datos a enviar.

  - mti_write( ).
  - mti_buscar_datosenviar( ).
  - mti_borrar_datoenviado( ).

- Mti_abort.c: Contiene tanto la función que aborta una conexión como la que la resetea.

  - mti_abort( ).
  - mti_reset( ).

- Mti_close.c: Almacena la función que cierra, ordenadamente, la conexión.

  - mti_close( ).

- Mti_enviar_cabecera.c: Contiene las funciones que crean la cabecera de la trama y la envían por el puerto serie del microcontrolador.

  - mti_enviar_trama( ).
  - mti_enviar_cabecera( ).
  - mti_convierte_formatoSCI( ).

- Mti_trama_completa_rx.c: La función que contiene es la encargada de esperar hasta que le llega una trama completa.

  - mti_trama_completa_rx( ).

- Mti_ack_fin.c: Contiene todas las funciones que se encargan de enviar una trama sin datos, sólo paquetes de ACK o FIN.

  - mti_enviar_ack( ).
  - mti_enviar_fin( ).
  - mti_enviar_finack( ).

- Mti_checksum.c: Almacena las funciones encargadas de calcular el checksum, tanto a la cabecera IP como a TCP incluyendo la pseudocabecera y los datos.

  - mti_ckecksum( ).
  - mti_ipchksum( ).
  - mti_tcpchksum( ).

- Mti_malloc_free.c: Contiene las funciones encargadas de almacenar o liberar memoria (estructuras).

  - mti_malloc_conex( ).
  - mti_free_conex( ).

- mti_malloc_enviar( ).
- mti_free_enviar( ).

- Mti_actualizar_nxt.c: Contiene las funciones necesarias para actualizar el valor del número de secuencia esperado y el número de acuse de recibo.

  - mti_actualizar_rcv_nxt( ).
  - mti_actualizar_ack_nxt( ).

- Mti_guarda_trama.c: La función que contiene este fichero es la encargada de almacenar la trama que se recibe si no es para la conexión que esta leyendo en ese momento, y almacenar la trama que se envía por si hubiera que volver a retransmitirla.

  - mti_guardar_trama( ).
  - mti_trama_anter( ).

- Mti_otros.c: Contiene funciones encargadas de recorrer las listas, tanto la que almacena la información de las conexiones activas como la que almacena los datos a enviar.

  - mti_buscar_tcpestadobanderas( ).
  - mti_buscar_conex_libre( ).
  - mti_buscar_id_conex( ).
  - mti_buscar_puerto( ).
  - mti_borrar_estructura( ).

- Mti.h: En este fichero se declaran todas las funciones de primer nivel, las que puede usar el usuario. También se definen los estados posibles que puede haber en TCP. Otras definiciones que se hacen son los valores que puede tomar el campo banderas en la cabecera TCP.

  En este fichero se definirán las estructuras necesarias como son:

  - MTI_ESTRUCTURA_CONEX: Estructura que contiene toda la información necesaria para una conexión. Posteriormente se definirá más detenidamente.

- MTI_TCPIP_CAB: Estructura que contiene todos los campos tanto de la cabecera TCP como la IP.
- SEND: Estructura donde se almacenarán los datos que se quiere mandar por la red. Almacenará información como el tamaño de los datos y el destino de éstos.

- Proces.h: Almacenará las definiciones de otras constantes necesarias en el proceso, declarará el prototipo de las funciones que no aparecen en mti.h así como las variables globales que son externas.

4.2 ESTRUCTURAS, CONSTANTES Y VARIABLES.

4.2.1 TIPOS

Se van a definir las siguientes tipos:

- unsigned char INT8U: Palabra de ocho bits, o lo que es lo mismo, un byte.
- unsigned short INT16U: Equivale a 2 bytes.
- unsigned long int u4: Definirá una palabra de 4 bytes.

4.2.2 ESTRUCTURAS CREADAS:

- MTI_ESTRUCTURA_CONEX: Estructura que almacenará toda la información necesaria para cada conexión. Contendrá los siguientes campos:
  - int num_conex: Número identificador de la conexión utilizada.
  - int espera_close: Cuando el proceso está en el estado CLOSE_WAIT y no hay más datos para enviar no se transmite nada, se queda a la espera de recibir la orden close del host remoto. Hasta que ello ocurra este campo estará activo para que no se envíen más tramas. Se explica mejor en la función *mti_read( ).*
  - int enviar_fin: Se activa cuando se llama a *mti_close( ).* Permanecerá en este estado mientras se esté esperando cerrar la conexión pero haya datos que enviar. Una vez que todos sean transmitidos se enviará un paquete de fin y este campo se desactivará.
  - int espera_respuesta: Informa acerca de si la trama que se ha enviado necesita asentimiento o no. Ello es debido a que existen tramas que no necesitan respuesta como son las tramas de asentimiento que no transportan datos.

- int enviar_ack: Se activa cuando es necesario enviar un asentimiento al host remoto porque la trama que ha llegado transporta datos que hay que asentir.
- INT8U tcpestadobanderas: indica el estado en el que se encuentra la conexión (LISTEN, CLOSED…)
- INT16U lc_puerto: número del puerto local utilizado en la conexión.
- INT16U rm_puerto: número del puerto remoto al que se va ha conectado.
- INT8U lc_ip_dir[4]: dirección IP local.
- INT8U rm_ip_dir[4]: dirección IP del par remoto.
- INT8U snd_nxt[4]: número de secuencia que envío.
- INT8U ack_nxt[4]: número de secuencia que se espera recibir y asentir.
- INT8U rcv_nxt[4]: número de acuse de recibo.
- INT8U temp: temporizador de retransmisiones. Indicará el número máximo de veces que puedo retransmitir un paquete.
- INT8U cont_retrans: contador del número de retransmisiones para un segmento particular.
- int temporizador1: Temporizador para controlar si ha llegado respuesta a la trama enviada antes de un determinado tiempo. Si transcurrido este tiempo no ha llegado la respuesta se transmitirá de nuevo la trama.
- int temporizador2: Cuando el proceso se encuentra en el estado "time_wait", se tiene que esperar un determinado tiempo en este estado para asegurar que la trama llega al otro extremo. Con este temporizador se controlará este tiempo de espera.
- struct mti_estructura_conex * proximo: Puntero a una estructura para poder tener una lista dinámica.

- MTI_TCPIP_CAB: estructura que contiene todos los campos tanto de la cabecera IP como la TCP.
  - Cabecera IP:
    - INT8U vhl: Versión y longitud de la cabecera IP
    - INT8U tipo: Tipo del servicio.
    - INT8U tam[2]: Tamaño total del datagrama.
    - INT8U ipid[2]: identificador. Tiene sentido cuando hay fragmentación.
    - INT8U ipoffset[2]: Desplazamiento del segmento si existe fragmentación.

- INT8U ttl: Tiempo de vida.
- INT8U proto: tipo de protocolo.
- INT16U ipchksum: Checksum de la cabecera IP.
- INT8U fuente_ip_dir[4]: Dirección IP de la fuente.
- INT8U dest_ip_dir[4]: Dirección IP del destino.

o Cabecera TCP:

- INT16U puerto_fuente: Número de puerto origen.
- INT16U puerto_dest: Número de puerto destino.
- INT8U seqno[4]: Número de secuencia.
- INT8U ackno[4]: Número de acuse de recibo.
- INT8U tcpoffset: Posición de los datos. Conociendo este número se conoce donde empezarían los datos si existiera el campo opciones.
- INT8U banderas: Bits de control. Determina el propósito del segmento.
- INT8U wnd[2]: Tamaño de la ventana.
- INT16U tcpchksum: Suma de control de la cabecera TCP, los datos y una pseudocabecera.
- INT16U urgp: Puntero urgente. Tiene sentido si hay datos urgentes.
- INT8U mti_datos[125]: información que transporta el datagrama.

- SEND: contiene la información necesaria para poder enviar un dato:
  - o int num_conex: Número de la conexión a la que se quiere enviar el dato.
  - o Int enviado: Se activará cuando la trama haya sido transmitida correctamente.
  - o int tam: Tamaño del dato a enviar.
  - o INT8U info[125]: Información que se quiere enviar.
  - o struct send * proximo: Puntero a la próxima estructura con el siguiente dato a enviar. Así se podrá crear una lista dinámica.

4.2.3 DEFINICIONES:

Se van a definir las constantes que se van a utilizar.

- Todos los posibles estados que pueden existir en TCP tendrán asociados un número y serán los siguientes:
  - CLOSED 0: La conexión esta cerrada.
  - SYN_RCVD 1: Se ha recibido un paquete con el bit SYN activo.
  - SYN_SENT 2: Se ha enviado un paquete con el bit SYN activo.
  - ESTABLISHED 3: La conexión está establecida.
  - FIN_WAIT_1 4: Se ha recibido una orden de cerrar.
  - FIN_WAIT_2 5: Se está esperando para cerrar la conexión.
  - CLOSING 6: Se está cerrando la conexión.
  - TIME_WAIT 7: En espera durante un determinado para cerrar la conexión.
  - LAST_ACK 8: En espera de acuse de recibo de la solicitud de finalización de la conexión.
  - LISTEN 9: El puerto está a la escucha.
  - CLOSE_WAIT 10: En espera de cierre por parte del usuario local.

- El campo bandera de la cabecera TCP llevará una de las siguientes etiquetas dependiendo del tipo de paquete que se esté enviando. Llevan el prefijo TCP porque representan las banderas de la cabecera del mismo nombre.
  - TCP_FIN 0x01: Paquete de fin de conexión.
  - TCP_SYN 0x02: Paquete de apertura de conexión.
  - TCP_RST 0x04: Paquete de reset.
  - TCP_PSH 0x08: Paquete con entrega de datos inmediata.
  - TCP_ACK 0x10: Paquete de asentimiento.
  - TCP_URG 0x20: El paquete lleva datos urgentes.

- Cuando llega un dato hay que comprobar si el asentimiento y el número de secuencia han llegado bien o no. Si sucede lo segundo habrá que retransmitirlo otra vez. Esto lo indicarán las siguientes definiciones:

  - mti_mal -1: El dato ha llegado mal.

  - mti_bien 1: El dato ha llegado bien.

- A la hora de establecer una conexión se necesita saber que números se pueden utilizar. Habrá que conocer si un número determinado esta libre u ocupado:

  - mti_libre 0: El número buscado está libre, se pude utilizar para una nueva conexión.

  - mti_ocupado 1: El número está ocupado por una conexión activa.

- La siguiente definición indica el número máximo de conexiones que pueden existir. Se ha considerado que un máximo de cinco conexiones abiertas es un buen número ya que al ocupar cada conexión una determinada memoria y ésta es limitada, por ello, si se tuvieran más conexiones el programa no sería eficiente.

  - mti_num_max_conex 5

- Habrá que saber si el protocolo por encima de IP es TCP o ICMP.

  - IP_PROTO_ICMP 1: El protocolo es ICMP.

  - IP_PROTO_TCP 6: El protocolo es TCP.

- Cuando un dato que se recibe se comprueba que esta mal hay que retransmitirlo, pero no se hará más de MTI_RTO veces. Se a considerado que tres es un buen número de retransmisiones ya que si después de tres veces no ha llegado el dato correcto es porque el problema está en la red y aunque se retransmitiera más veces seguiría dando error.

  - MTI_RTO 3

- El tamaño del buffer de entrada de datos por el puerto serie asíncrono viene definido por

  - TAM_BUFFER1 256

- Cuando se transmiten o se reciben datos hay que realizarlo a una velocidad determinada. Se definen dos constantes con las dos posibles velocidades que se pueden utilizar.

- Baud_9600 0x00c0
- Baud_115200 0x0010

- En un programa se puede tener la posibilidad de habilitar o deshabilitar un periférico. Se harán las siguientes definiciones:
  - ENABLE 1: El periférico está habilitado.
  - DISABLE 0: El periférico está deshabilitado.

- Otra posibilidad que existe es la de habilitar una interrupción asociada a un periférico o no habilitarla. Para ello se encuentra:
  - ENABLE_INT 1: Interrupción habilitada.
  - DISABLE_INT 0: Interrupción deshabilitada.

4.2.4 VARIABLES GLOBALES:

A continuación se detallan todas las variables globales que se van a utilizar en todo el programa, es decir, variables que pueden ser modificadas desde cualquier punto de éste.

- int mti_num_conexión[mti_num_max_conex]: Array que informa del estado de las conexiones. Indica si están ocupadas o no y si podrían ser utilizadas cuando se quiera establecer una nueva conexión.
- INT16U mti_tam_dato: Contiene la longitud del dato que se esté transmitiendo por la conexión que esté activa en ese momento.
- MTI_ESTRUCTURA_CONEX * mti_conex: Apuntará a la estructura que contiene la información de la conexión con la que se esté trabajando en un determinado momento.
- MTI_ESTRUCTURA_CONEX * mti_conexion: Puntero a la primera estructura de la lista de conexiones activas.
- MTI_ESTRUCTURA_CONEX mti_conexion1: Estructuras donde se almacenará la información de cada una de las conexiones abiertas en un determinado momento (mti_conexion1,…, mti_conexion5). Se definirán cinco estructuras para cinco posibles conexiones.
- SEND * mti_datos_a_enviar: Apuntará al principio de la lista que contiene todos los datos a enviar.
- SEND mti_datos_a_enviar1: Estructuras donde se almacenarán los datos que se pretenden enviar. Se han definido cinco estructuras (mti_datos_a_enviar1,…, mti_datos_a_enviar5) para reservar datos ya que la memoria es un recurso limitado, si se decidiera tener más datos a la espera se

perdería eficiencia ya que se estaría utilizando mucha memoria y, además, si el programa funciona correctamente, da tiempo a enviar todos los datos pendientes.

- static INT8U inic_numsec[4]: Número de secuencia inicial.
- static INT16U mti_ultimo_puerto: Guarda la información del último puerto utilizado para facilitar la labor a la hora de buscar el siguiente puerto sin usar en la próxima conexión.
- INT8U MTI_IP_DIR0, INT8U MTI_IP_DIR1, INT8U MTI_IP_DIR2, INT8U MTI_IP_DIR3: Almacenará la dirección IP local.
- static int mti_num_listen: Número que se le asigna a un puerto cuando se pone a la escucha y pasa al estado LISTEN para no confundirlo con otro que ya esté a la escucha.
- MTI_TCPIP_CAB *MTI_BUF: Puntero a la trama que se tiene que procesar. Una vez tratada apuntará a la trama que se crea para ser enviar al host remoto.
- MTI_TCPIP_CAB mti_trama_anterior: Almacena la trama enviada por si hubiera que volver a reenviarla.
- MTI_TCPIP_CAB mti_BUFFER_PAQUETES[3]: Array para almacenar las tramas que llegan para una determinada dirección IP pero que no van destinadas al puerto de la conexión que está leyendo en ese momento.
- INT8U buffer1[TAM_BUFFER1]: Buffer donde se almacenan los datos que van llegando por el puerto serie asíncrono.
- INT16U first1: Apunta al ultimo elemento que se ha almacenado en el buffer1 más uno, es decir, apunta al primer elemento en el que se puede almacenar un dato.
- INT16U last1: Apunta al primer elemento que se almacenó en el buffer1. Será el primer elemento en salir cuando se quieran sacar datos del buffer1.

4.3 CÓDIGO DE LAS FUNCIONES CREADAS:

4.3.1 FUNCIONES DE PRIMER NIVEL

- ***void mcore_init_system( )***: Función que inicializa el microcontrolador. No se le pasa ningún parámetro ni tampoco devuelve ningún valor.

  Inicialmente se llamará a la función *SystemInit( )* que será la encargada de inicializar todos los registros del MCORE.

  A continuación se llamará a las rutinas que configuran los periféricos. En este proyecto se necesitarán los temporizadores y el puerto serie asíncrono que se configurarán de la siguiente manera:

  - *SCI1(Baud_115200,ON_DOZE,ENABLE,ENABLE)*: Configura el puerto serie asíncrono a la velocidad de 115200 baudios y habilitando tanto la transmisión como la recepción.
  - *PIT1(4,ON_DOZE,ENABLE_INT,ENABLE,10000)*: Configura el temporizador habilitando la interrupción.

  Por último habilitará las interrupciones de prioridad 1 y prioridad 2 que son las que se necesitan en el programa.

  - mmc_enNormalInterrupt1
  - mmc_enNormalInterrupt2
  - EnableInterrupts

- ***void mti_init (INT8U local_addr[ ])***: Función que inicializa el sistema. Se le pasa como parámetro la dirección IP que se quiere inicializar.

  En primer lugar se llamará a la función *mcore_init_system( )* que inicializará al microcontrolador.

  A continuación se inicializarán a libres todos los elementos del array mti_num_conexión[] para poder disponer de todas las conexiones posibles.

  Se hará apuntar tanto mti_conexion como mti_datos_a_enviar a NULL ya que no existen ni conexiones activas ni datos para enviar.
  Se inicializará la variable que contiene la información del último puerto utilizado (*mti_ultimo_puerto*) a cero.
  El numero que identifica a un puerto que se pone a la escucha (*mti_num_listen*) se inicializará al número máximo de conexiones posibles más uno para que no se confunda un puerto que está a la escucha con una conexión activa.
  El campo num_conex de las estructuras mti_datos_a_enviar1,…, mti_datos_a_enviar5, mti_datos_anviar1,…, mti_datos_a_enviar5 se

inicializará a –1 para indicar que las estructuras no están siendo utilizadas ni por una conexión ni por datos para enviar, respectivamente:

- mti_datos_a_enviar1.num_conex = -1;
- mti_conexion1.num_conex = -1;

Con todo esto ya se tiene inicializado el sistema.

- ***int mti_listen (INT8U local_addr[ ], INT16U local_port)***: El objetivo de esta función es poner a la escucha un puerto para una posible conexión. Se le pasará como parámetros el puerto en cuestión y la dirección IP. Devolverá –1 si se produce algún error.

  Los pasos que se van a seguir son los siguientes:

  1. Se busca si el puerto que se quiere poner a la escucha ya está siendo usado. La función utilizada ello es mti*_buscar_puerto(local_port).*
  2. Si el puerto no está siendo usado se comprueba que no esté a la escucha.
  3. Si el puerto buscado no está siendo utilizado y se ha comprobado que no estaba puesto a la escucha, hay que ponerlo.

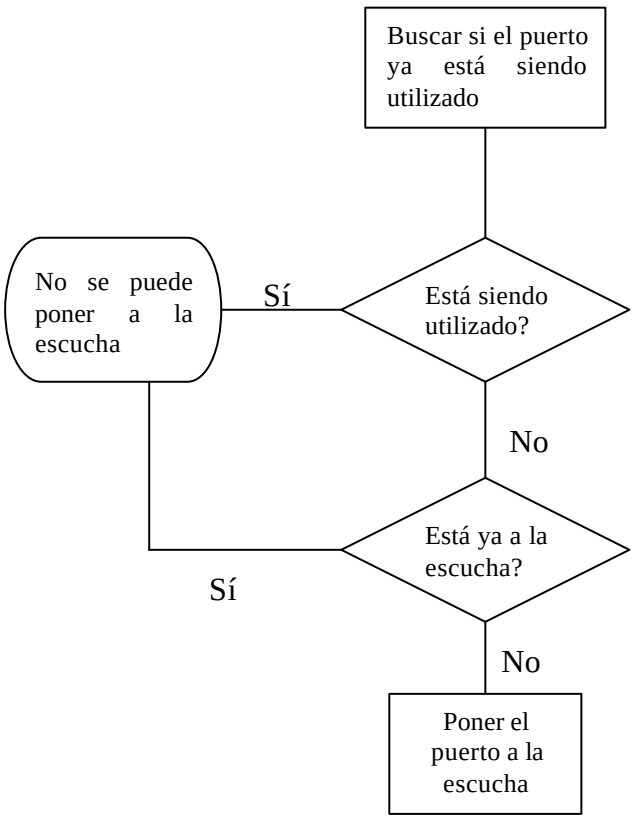

Figura 4.1

Cómo poner un puerto a la escucha:

1. Se reservará espacio para una estructura que mantendrá la información de qué puerto se ha puesto a la escucha, la dirección IP y el estado (LISTEN). Si aún no hay ninguna estructura creada, se hará apuntar *mti_conexión*(puntero a las estructuras que mantienen información de las conexiones) a *mti_conexion1* y aquí se almacenará la información. Si ya hay estructuras creadas se llamará a la función *mti_malloc_conex()* para buscar una estructura que esté libre. Se recorrerá la lista hasta que se encuentre la última estructura utilizada para conectar la nueva estructura a ella. Si no hubiera alguna estructura libre o existiera algún error la función devolvería –1.
2. Una vez creada la estructura se actualizan los campos de los que se dispone información como son:
   - tcpestadobanderas: Estado de las banderas actualizado a LISTEN
   - lc_ip_dir: Dirección IP. Se actualizará con el valor que se le pasa como parámetro (*local_addr[]*).
   - lc_puerto: Puerto local. Se actualizará con el valor que se le pasa como parámetro (*local_port*).
   - El campo num_conex se actualizará a mti_num_listen. Éste valor no es el del número de conexión sino un valor auxiliar para poder identificar que el puerto está a la escucha. Cuando se acepte la conexión, este campo será actualizado por el verdadero valor del identificador de la conexión.
   - Mti_num_listen se actualizará sumándole uno para que en la próxima llamada que se realice a la función mit_listen no exista ningún error.
   - El campo enviar_fin, que mantiene información acerca de si se a realizado una llamada para cerrar la conexión, se inicializará a mti_no. Se activará cuando se realice una llamada a *mti_close*
   - El campo espera_close se inicializará a mti_no ya que sólo se activará cuado el host remoto envíe un paquete de FIN.
   - Si todo ha ido bien, la función devolverá el número identificador del puerto que está a al escucha: puntero->num_conex.

- ***int mti_connect(INT8U rem_addr[], INT16U rm_puerto)***: Es llamada cuando un host se quiere conectar a un par remoto. Se le pasa como parámetros tanto la dirección IP del host remoto como la del puerto al que se desea conectar. Devolverá un identificador de la conexión si se puede conectar correctamente o –1 si ha sucedido algún error.

  Se va a buscar un puerto sin usar para asignarlo a esta conexión. Hay que diferenciar varios casos:

- Caso 1: Caso en el que no existe ninguna conexión abierta, es decir mti_conexion apunta a NULL ya que no hay ninguna estructura en la lista. Se buscará un número de conexión libre, para ello se utiliza la función *buscar_conex_libre().* Si el número encontrado es -1, existe algún error en la búsqueda de una conexión libre. Si el valor devuelto es distinto de –1 la búsqueda ha sido exitosa. Como no existe ninguna conexión activa *mti_conexión* tendrá que apuntar a *mti_conexion1* donde se almacenará toda la información necesaria para esta conexión. Una vez que se tiene una estructura reservada para esta conexión se llamará a la función *mti_encontrada_conexión()* que se encargará de actualizar toda la información que necesita la conexión.

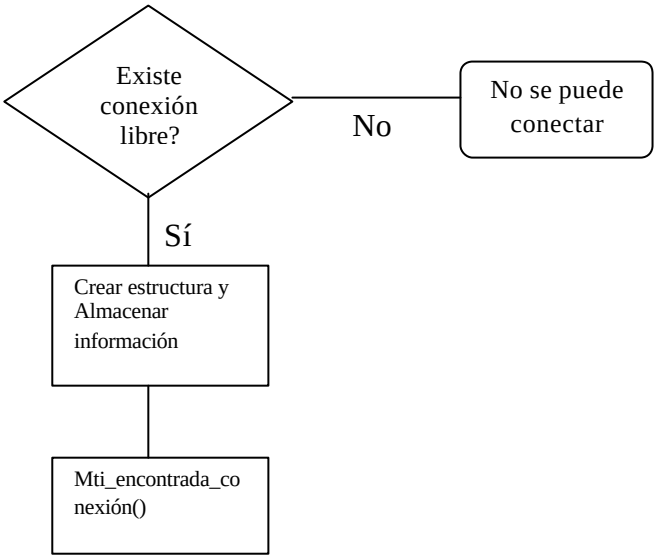

Figura 4.2

- Caso 2: El puerto almacenado en *mti_ultimo_puerto* ya está siendo utilizado. Se recorrerá la lista de conexiones activas aumentando el valor de éste hasta que se encuentre un número de puerto que no esté siendo utilizado. Una vez encontrado se pasará a los siguientes casos para seguir las comprobaciones.

  - Caso 3: Se busca alguna conexión que esté en el estado TIME_WAIT. Ello significa que esa conexión se ha cerrado y por tanto se puede utilizar su estructura para almacenar la información de la nueva conexión. Ya se tiene reservada la estructura necesaria y por ello se llamará a la función *mti_encontrada_conexión()* para actualizar toda la información que necesita la conexión. El puerto local será el almacenado la variable *mti_ultimo_puerto.*

  - Caso 4: Si no se ha encontrado una conexión en el estado TIME_WAIT, se tendrá que crear una estructura para la conexión que se quiere abrir, donde el puerto local es el almacenado en

mti_ultimo_puerto. Al igual que en el caso 1, primero se buscará una conexión libre. Tras encontrarla se reservará memoria para dicha estructura con la función *mti_malloc_conex()* y se recorrerá la lista de conexiones activas para encadenar esta estructura a la última encontrada. Por último se llamará a *mti_encontrada_conexión()* para actualizar todos los campos.

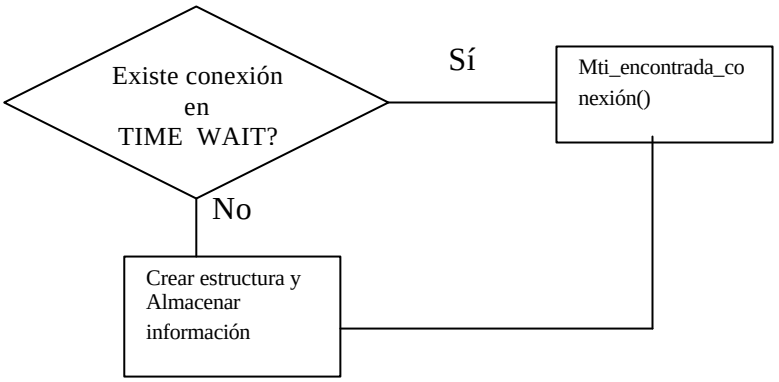

Figura 4.3

Si todo este proceso se ha realizado con éxito, la función devolverá el identificador de la conexión que se ha abierto.

- ***int mti_accept (int x)***: Acepta una conexión cuando llega un paquete de SYN. Devuelve -1 si hay error. Se le pasa como parámetro el número que dio la función *mti_listen* tras poner el puerto que se desea que acepte la conexión a la escucha. Si dicho número fuera –1 existió algún error en la puesta a la escucha del puerto y, por tanto, esta función también devolverá un valor -1. Si no sucede esto, se continúa con el proceso de aceptación.

  En primer lugar se tendrá que comprobar si el puerto sigue a la escucha, es decir, si ninguna otra conexión ya lo está utilizando:

  1. Primer se busca la estructura que mantiene toda la información del puerto que está a la escucha. Para ello se utiliza la función *mti_buscar_id_conex().* Si no encontrara dicha estructura, la función habría terminado su proceso y devolvería –1. Si sí encuentra la estructura que se busca, se hará que *mti_conex* apunte a ésta para trabajar directamente con *mti_conex*.
  2. En segundo lugar se comprobará si existe alguna otra conexión que se haya adelantado a ésta y haya utilizado este puerto antes. Si este es el caso m*ti_accept()* tendrá que devolver –1 ya que el puerto ya está siendo utilizado. Para realizar este proceso se utilizará la función *mti_buscar_puerto* para buscar el puerto deseado y se comprobará en el estado en el que se encuentra.

  Una vez que se ha comprobado que el puerto sigue estando a la escucha hay que esperar hasta recibir un paquete para seguir con el proceso:

1. En un primer paso se buscará en el array *mti_BUFFER_PAQUETES* si existe algún paquete almacenado que sea para esta conexión. Si se encuentra dicho paquete, se almacena en *MTI_BUF* para utilizar la información que contiene en las sucesivas funciones.
2. Si no hay ningún paquete almacenado para esta conexión, se tendrá que esperar hasta que llegue un paquete por el buffer de entrada del microcontrolador. Se llamará a la función *mti_trama_completa_rx()* que se encargará de recibir una trama completa. Mientras se está recibiendo la trama la función anterior devolverá un cero para que la función *mti_accept()* sepa que aún no ha llegado una trama completa. Si se produce algún error, como puede ser un checksum erróneo, devolverá –1 y si no es un paquete para su dirección IP devolverá 2. Cuando se pone a esperar que llegue una trama se activa un temporizador para que, si transcurrido un determinado tiempo no ha llegado la trama, se devuelva error ya que se ha perdido el paquete que transportaba la información del host que se quería conectar. Si la trama que llega no es para el puerto que estaba a la escucha se guardará en el array mti_BUFFER_PAQUETE para que otra conexión la pueda utilizar. Para ello se llamará a la función *mti_guardar_trama(). mti_accept* devolverá 3 indicando que la trama que ha leído no es para esta conexión.

Una vez que ya se tiene el paquete para la conexión deseada se comprueba el campo banderas. Si es distinto de SYN es un error ya que para aceptar una conexión ésta es la bandera que tiene que llegar. Se devolverá –1 indicando que no se ha podido aceptar la conexión.

El siguiente paso será buscar una conexión libre para poder conectarse y almacenar toda la información. Si no se encuentra se devolverá error. Una vez encontrada se actualizan los campos de la estructura:

- num_conex será el valor devuelto por *buscar_conex_libre()*.
- Temp: Se actualizará al valor máximo posible de retransmisiones.
- El contador de retransmisiones empezará en cero.
- El estado de esta conexión será SYN_RCVD ya que le ha llegado un SYN y se tiene que enviar un SYNACK que se actualizará en *MTI_BUF->banderas = TCP_SYN | TCP_ACK* .
- Se actualiza el puerto remoto al valor de *MTI_BUF->puerto_fuente.*
- La dirección IP remota se actualizará al valor dado por *MTI_BUF->fuente_ip_dir*.
- El número de secuencia se actualizará con inic_numsec.
- ack_nxt será el valor anterior más uno y rcv_nxt es actualizará con BUF->seqno +1 que será el número de acuse de recibo deseado según se explicó en el protocolo.

Por último se llamará a *mti_enviar_trama()* para enviar el paquete SYNACK.

- ***int mti_write(int socket, INT8U * mensaje,INT16U tam_mensaje):*** Almacena en una lista los datos que se quieren enviar al host remoto. Cuando se cree un paquete, se buscará en esta lista datos pendientes de ser enviados y si los hubiera se incluirían en el paquete.

  Se le pasa como parámetros el número de conexión al que se quiere enviar la información, los datos y el tamaño de éstos.

  Los pasos a seguir son los siguientes:

  1. Paso 1: Con la función *mti_buscar_id_conex(socket)* se busca si existe la conexión a la que se quieren enviar los datos. Si no existiera se devolvería error (–1) ya que no se puede enviar información a una conexión inexistente.
  2. Paso 2: Si se hubiera hecho una llamada a *mti_close()* con el fin de cerrar la conexión, no se podrán escribir más datos y la función deberá devolver error. Para ello se observa la variable enviar_fin de la estructura de información. Esta variable se activa cuando se ha realizado una llamada a *mti_close(),* indicando que se quiere cerrar la conexión. Si al pretender escribir datos en la lista se observa que este campo está activo, se devolverá error. Si no, se seguirá con el procedimiento.
  3. Paso 3: Si el estado de la conexión es LISTEN se devuelve –1 ya que aún no existe la conexión.
  4. Paso 4: Una vez que se ha comprobado que todo es correcto, se procede a almacenar el dato en la lista. Pueden suceder varios casos:
     - Aún no existe ningún dato almacenado en la lista. El puntero mti_datos_a_enviar se hará apuntar a la primera estructura que puede tener la lista (mti_datos_a_enviar1) y se actualizarán todos los campos de esta con los parámetros que se le pasaron a la función, es decir, con el número de conexión, los datos y el tamaño de éstos. El puntero a la próxima estructura apuntará a NULL.
     - Si ya existen en la lista datos pendientes de envío, se buscará la última estructura encadenada y se llamará a *mti_malloc_enviar()* para reservar una estructura donde poder almacenar la información. Si esta función devolviera NULL es porque no se ha encontrado una estructura libre que se pueda utilizar y, por tanto, se devolvería error. Si sí se ha encontrado, la función dará el puntero a la estructura encontrada y se procederá a actualizar todos los campos como en el caso anterior.

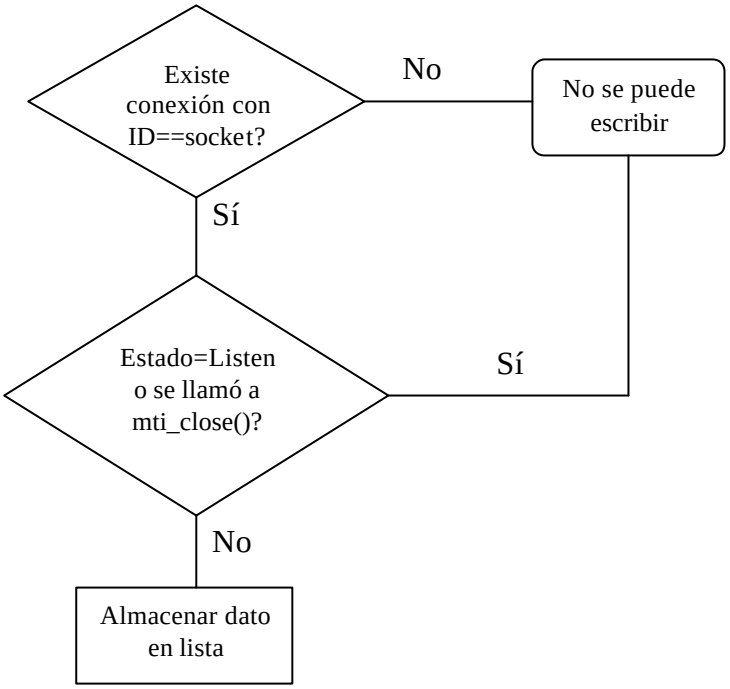

Figura 4.4

- ***int mti_read(int socket,INT8U *buffer,int *tam_buffer)***: Función encargada de leer los paquetes que llegan por un buffer de entrada y procesarlos. Para ello estudiará si el paquete es para esa conexión, si los números de asentimiento y secuencia que transporta son los esperados, si el checksum es correcto,…. En función de esta información responderá con un determinado paquete o devolviendo –1 si se detectara algún error.

  Los parámetros que se le pasará serán los siguientes:

  - socket: Número de la conexión de la que se quieren leer los datos.
  - buffer[]: Array en el que se almacenarán los datos procedentes del host remoto. Si el usuario quisiera leer los datos que le han enviado desde el par remoto, tendrá que leer este buffer.
  - tam_buffer: variable que almacena el tamaño del dato que hay en buffer. Si este valor fuera cero, significaría que el par no ha enviado datos y, por tanto, el buffer estará vacío. El valor se pasa por referencia para que pueda ser modificado por la función.

  Se procede a continuación a detallar esta función.

  1. En primer lugar hay que buscar si existe la conexión de la que se quieren leer los datos. Para ello, con la función *mti_buscar_id_conex(socket),* se buscará una estructura que contenga información de una conexión activa cuyo identificador sea el mismo que el que se le pasa como parámetro y devolverá un puntero a la estructura encontrada.

- Si el puntero apunta a NULL, no se habrá podido encontrar la conexión buscada y la función devolverá error(-1) .

- Si apunta a una estructura, ésta será la que tendrá toda la información de la conexión con la que se quiere trabajar y, por tanto, habrá que hacer que mti_conex (puntero a la conexión con la que se trabaja en un determinado momento) apunte a estructura devuelta por *mti_buscar_id_conex(socket).*
  - ✓ mti_conex=mti_buscar_id_conex(socket);

2. Se tiene que esperar hasta que llegue una trama para esta conexión. MTI_BUF apuntará a dicha trama.
   Hay que recordar que MTI_BUF actúa de buffer de entrada cuando llega un paquete y de buffer de salida cuando se crea una trama. En esta función primero tendrá que almacenar la información del paquete que se quiere procesar.

   Para esperar una trama se pueden dar dos casos:

   - Caso 1: Se busca la trama en el buffer que almacena los paquetes que han llegado y nadie a utilizado. Para ello, MTI_BUFF irá apuntando a cada uno de los elementos del array mti_BUFFER_PAQUETES para buscar si existe alguna trama cuyo puerto destino sea igual al puerto local de esta conexión (MTI_BUF->puerto_dest==mti_conex->lc_puerto) ya que, si ello sucediera, esta sería la trama que se estaba esperando. Si no se encuentra ninguna, se pasaría al caso 2.

   - Caso 2: No existe ninguna trama almacenada que sea para esta conexión así que habrá que esperar hasta que llegue una nueva trama. Para ello:

     - El temporizador1 se inicializa a cero.

     - Se llama a *mti_trama_completa.* Esta función espera hasta que llegue una trama completa y comprueba si trae errores (devuelve –1). Mientras no llegue la trama completa devolverá 0. Cuando ya le haya llegado la trama completa y haya comprobado que es correcta devolverá 1.

     - Se entra en un bucle mientras que *mti_trama_completa* no devuelva un 1, es decir, mientras que no llegue toda la trama. Si transcurrido un determinado tiempo no ha llegado todo el paquete, se tendrá que retransmitir la trama anterior ya que se ha podido perder por el camino. Si la trama ya se ha retransmitido MTI_RTO, no se podrá retransmitir más (si no se pone un número máximo de retransmisiones podría existir algún problema en el otro extremo de la conexión, y estar retransmitiendo indefinidamente). Esto se realiza de la siguiente forma:

i. Se comprobará si temporizador1 (lo va incrementando el modulo temporizador1 del microcontrolador) ha superado un determinado valor (se ha configurado para que cuente hasta 1.25 segundos ya que es un tiempo suficiente para recibir una trama. Se podría poner un número menor y el proceso sería más rápido):

   (mti_conex->temporizador1>=1000)

ii. Si el contador no ha superado el tiempo máximo se seguirá con el proceso.

iii. Si el contador a superado el tiempo máximo se pueden dar varios casos:

   a) No se espera asentimiento de la trama que se envió. Se buscará si existen datos para enviar a esta conexión. Si sí existen se llamará a *mti_proceso_estados()* y se enviarán. Si no existen se retornará.

   b) Si sí se esperaba asentimiento se comprobará el número de retransmisiones (mti_conex->cont_retrans>=mti_conex->temp). Si supera el máximo la función devolverá error. Si no lo supera, se volverá a retransmitir el paquete anterior y se incrementará el número de retransmisiones (mti_conex->cont_retrans=mti_conex->cont_retrans+1).

3. Si la función *mti_trama_completa()* ha devuelto –1, es porque ha existido un error y la función principal también tendrá que devolver error.
4. Si la función *mti_trama_completa()* ha devuelto 2 es porque la trama no es para la dirección IP local y la función principal devolverá 2.
5. Si el paquete ha llegado correctamente se comprueba si el paquete es para el puerto local de la conexión con la que se está trabajando. Si no es así, se tendrá que almacenar en mti_BUFFER_PAQUETES, para que otra conexión lo pueda utilizar. Se llamará a la función *mti_guardar_trama()* para guardar la trama que se ha recibido en el buffer. La función principal devolverá 3 indicando que la trama no es para esa conexión.
6. Una vez que ya se ha comprobado que el paquete es para esta conexión, se extraerán los datos que haya enviado el host remoto. Para ello se almacenarán en buffer y el tamaño en tam_buffer y el usuario ya los podrá utilizar.
7. Por último se llamará a la función *mti_proceso_estados( )*. Esta se encargará de hacer el estudio del estado el que se encuentra la conexión, a qué estado debe evolucionar y que tipo de trama crear. Si esta función devuelve –1 es que se ha producido un error y la función principal también deberá devolver error. Si la función devuelve 1 todo se ha realizado correctamente y se devolverá un 1.

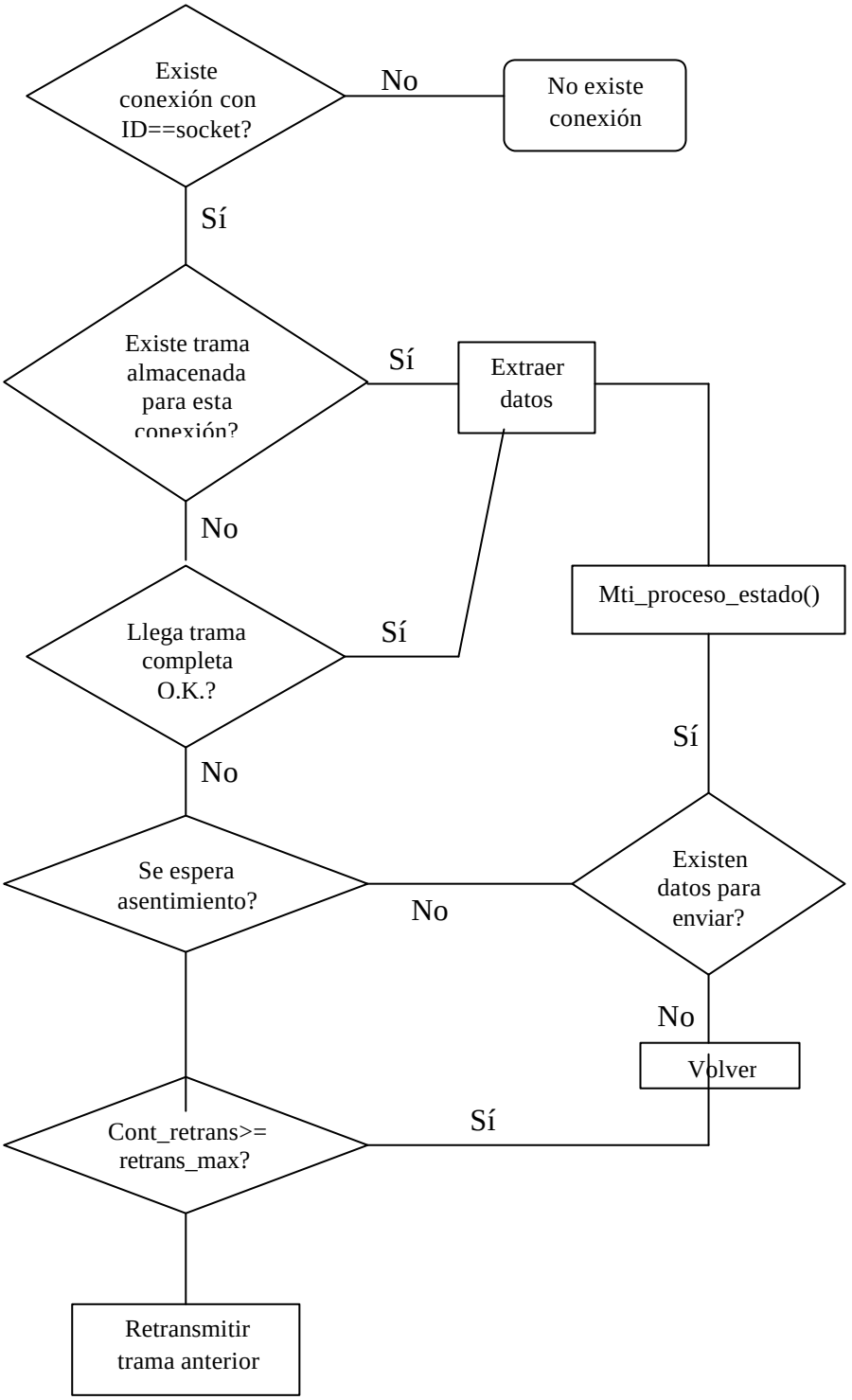

Figura 4.5

- ***int mti_close (int numero_de_conexion)***: Esta función debe ser llamada cuando se quiere cerrar la conexión ordenadamente. Una vez realizada la llamada, no se cierra inmediatamente, si no que habrá que enviar todos los datos pendientes para esa conexión. Si se quisiera enviar algún dato más utilizando la función *mti_write()* no se podría ya que sólo se podrán enviar los que hubieran sido puestos en la lista antes de la llamada a close.

Para cerrar definitivamente la conexión, se tendrá que esperar a que el host remoto también la cierre. Mientras tanto podrá aceptar los datos que éste le mande.

El procedimiento que seguirá será el siguiente:

1. Se buscará la conexión cuyo identificador sea el mismo que el valor que se pasa a esta función como parámetro. Para ello se utiliza la función *mti_buscar_id_conex(numero_de_conexion)*, la cual devolverá un puntero que podrá apuntar a:

   - NULL si, por cualquier motivo, la conexión no existiera y por tanto *mti_close()* devolverá error (-1) ya que no puede cerrar una conexión que no existe.

   - Estructura que mantiene la información de la conexión buscada. Se tendrá que estudiar el estado en el que se encuentra la conexión en ese momento (observando el estado de tcpestadobanderas en la estructura encontrada) y actuar devolviendo –1 si existe error o 1 si se puede cerrar:

     - Estado CLOSED: La conexión ya está cerrada y, por tanto, no se puede volver a cerrar. La función devolverá error (–1).
     - Estado SYN_SENT: La conexión estaba en proceso de conectarse y ya se ha creado la estructura con al información de la conexión, por tanto, habrá que borrar dicha estructura (llamando a la función *mti_borrar_estructura(punt_conex))* para que otra conexión la pueda utilizar y devolver 1 ya que sí se ha cerrado.
     - Estados SYN_RCVD, ESTABLISHED y CLOSE_WAIT: Activará la variable enviar_fin (*punt_conex->enviar_fin=mti_si*) para que, si no existieran datos pendientes de ser enviados se envíe un paquete de fin y si sí existieran se transmitan éstos y por último el paquete de fin. Este proceso se realiza en la función *mti_read()*. La función devolverá 1 ya que si se puede cerrar la conexión.
     - Estados FIN_WAIT_1, FIN_WAIT_2, CLOSING, LAST_ACK y TIME_WAIT: Cuando la conexión se encuentra en uno de estos estados es porque ya se estaba cerrando y, por tanto, no se podrá cerrar una conexión que ya está en ello. Así pues, la función devolverá error (-1).

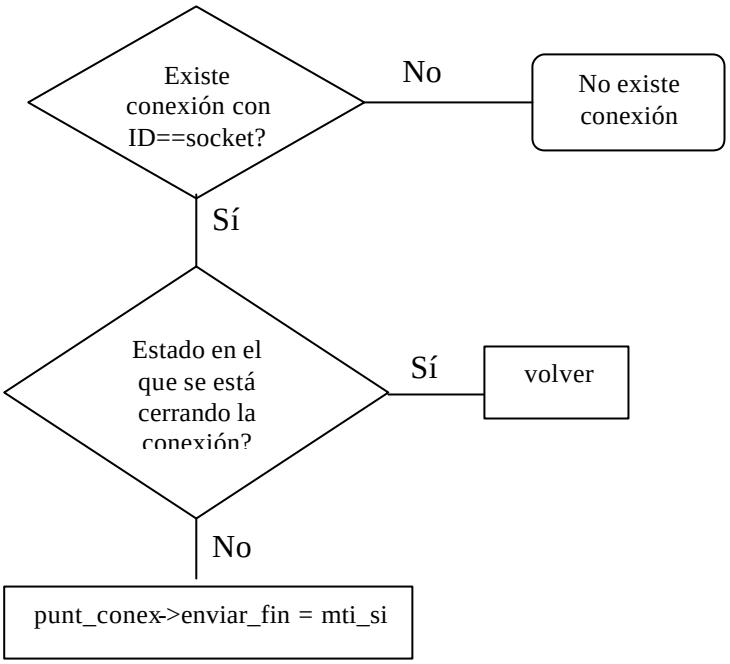

Figura 4.6

- ***int mti_abort (int numero_de_conexion)***: Aborta la conexión que indica numero_de_conexion. Si hay algún error devolverá -1.

  El procedimiento seguido será:

  1. En primer lugar se buscará con la función *buscar_id_conex(numero_de_conexion)* la estructura que contenga como identificador de conexión la conexión que se quiere abortar. Si no se encuentra se devolverá -1 ya que no se puede abortar una conexión que no existe.
  2. Una vez encontrada la conexión que se quiere abortar se comprueba en qué estado se encuentra.
     - Si está cerrada (CLOSED) se devuelve -1 ya que no se puede abortar una conexión que ya está cerrada.
     - Si está en LISTEN se borrará la estructura (mti_*borrar_estructura(punt_conex)*) pasa al estado CLOSED, pero no se envía nada al otro extremo ya que aún no se había establecido la conexión.
     - Si la conexión está en SYN_SENT (ha enviado un SYN) se tienen que borrar todos los datos que estuvieran en la cola para enviar a esta conexión: buscar si hay datos a enviar (*buscar_datoenviar(mti_conex->num_conex))*, si los hay se borra la estructura *(borrar_datoenviado(mti_conex->num_conex))*
     - Si la conexión se encuentra en SYN_RCVD, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2 o CLOSE_WAIT primero se borrarán todos los datos que estuvieran esperando para ser enviados a esta

conexión, como en el caso anterior. Tras esta operación se enviará un paquete de reset *(mti_reset())* y se borrará la estructura que identifica a esta conexión(*borrar_nodo(punt_conex)*).

- Si el estado de la conexión es CLOSING, LAST_ACK o TIME_WAIT como ya se está cerrando la conexión, se borrará el nodo que contiene toda la información de ésta (*mti_borrar_estructura(punt_conex)*) y todos los datos que hubiera en la cola par enviarle a esta conexión.

4.3.2 FUNCIONES DE SEGUNDO NIVEL

- ***Int mti_encontrada_conexion(MTI_ESTRUCTURA_CONEX * conex1, INT8U rem_addr[], INT16U rm_puerto)***: Se le pasarán como parámetros:

  - Un puntero a la estructura creada en la función *mti_connect()* y que será la que se querrá actualizar.

  - El puerto remoto al que se quiere conectar el host local.

  - La dirección remota a la que se conectará.

  Devolverá –1 si ha sucedido algún error y el número de la conexión si todo ha sucedido bien.

  Los pasos que se seguirán serán:

  1. Se actualizará el número de secuencia (*snd_nxt*) que se envía con el valor almacenado en *inic_numsec*. Éste será un número aleatorio.
  2. El número se secuencia que se espera recibir (*ack_nxt*) se actualiza al número de secuencia más uno ya que, como se comento al explicar el protocolo TCP (ver punto 4.2), al realizar la conexión se asiente un número más del enviado.
  3. El número de acuse de recibo (*rcv_nxt*) se inicializará a cero porque, hasta que no se reciba el paquete SYNACK, no se sabe cual es el número de inicio de secuencia del par.
  4. El valor del contador de retransmisiones (*cont_retrans*) se inicializará a cero ya que es la primera vez que se envía este paquete.
  5. El valor máximo posible de retransmisiones (variable *temp* de la estructura) será uno ya que el paquete de apertura de conexión sólo se enviará una vez.
  6. El puerto local se actualiza con *mti_ultimo_puerto.* El puerto remoto y la dirección IP remota se actualizarán con los parámetros que se le han pasado como argumentos a la función.

7. El estado en el que se encuentra la conexión en este momento es SYN_SENT y ello se reflejará en le campo *tcpestadobanderas*.
8. El campo espera_close se inicializará a mti_no ya que sólo se activará cuado el host remoto envíe un paquete de FIN.
9. El campo enviar_fin, que mantiene información acerca de si se a realizado una llamada para cerrar la conexión, se inicializará a mti_no. Se activará cuando se realice una llamada a *mti_close().*
10. En el buffer de salida se pondrá en el campo banderas TCP_SYN (*BUF->banderas = TCP_SYN*) ya que este es el paquete que se quiere enviar para establecer una conexión. *mti_conex()* apuntará a la conexión en uso.
11. Por último se llama a la función *mti_enviar_trama( )* que se encargará de mandar el paquete al par actualizando todos los campos necesarios.

La función devolverá a la función de nivel superior, *mti_connect(),* el número de conexión si todo se ha realizado correctamente

- ***int mti_proceso_estados( )***: Procesa el estado en el que se encuentra la conexión con la que se está trabajando en ese momento. Estudiará toda la información que trae la trama de entrada y, en función de esto, evolucionará a un determinado estado. A su vez, creará una trama de contestación a la anterior o con datos que transmitir.

  El procedimiento que se seguirá es:

  1. Si el paquete que se está procesando es un reset, quiere decir que el host remoto ha abortado la conexión y, por tanto, habrá que pasar al estado CLOSED, borrar todos los datos pendientes de envío para esa conexión y borrar la estructura que mantenía la información de la conexión. para ello se llamará a la función *mti_borrar_estructura(mti_conex)* y se devolverá 1.
  2. Si el número de asentimiento de la trama entrante es el esperado, el paquete que ha llegado está bien (MTI_BUF->ackno[] == mti_conex->ack_nxt[]). Se actualizará el número de secuencia y se inicializará el contador de retransmisiones ya que es la primera vez que se envía este paquete. Si el número de asentimiento no es el esperado, la trama no viene bien y habrá que retransmitir el paquete anterior (para ello se llama a la función *mti_fin_procesoestados( )*) ,y a su vez, incrementar el contador de retransmisiones. Si éste hubiera superado el número máximo de retransmisiones, no se volvería a retransmitir el paquete y la función devolvería error.
  3. También se comprobará si el número de secuencia que ha llegado era el que se esperaba (MTI_BUF->seqno[] == mti_conex->rcv_nxt[]).Si es correcto, se sigue con el procedimiento. Si hay algún error se llamará a *mti_enviar_ack()* para que se envíe un paquete ACK con el mismo número de secuencia que el último paquete se envió. Con ello se informa al otro extremo de la conexión que ha existido algún error y que vuelva a enviar el último paquete que envió.

A partir de este momento, se comprobará en qué estado se encuentra y como actuar:

4. Estado SYN_SENT: Se comprueba si las banderas que están activadas en el paquete procedente del host son el bit ACK y el bit SYN. Si no es así, no es un paquete de aceptación de conexión que es lo que se esperaba recibir. Ello indica que se ha sucedido un error. Si el paquete sí es el esperado se pasará al estado ESTABLISHED y se llamará a *mti_enviar_ack( )* ya que es necesario asentir este paquete para terminar el proceso de apertura de una conexión.
5. Estado SYN_RCVD: Se envió un paquete de SYNACK como respuesta a un paquete de SYN. Se esperará una trama de asentimiento para completar el establecimiento de la conexión. Por ello, se comprueba si el campo banderas de la trama que ha llegado lleva activo el bit ACK. Si es así la conexión a sido establecida pasando el estado a ESTABLISHED. Se llamará a la función *mti_fin_procesoestados( )* para que se cree la trama y se envíen datos el otro extremo (si los hubiera). Si el paquete que llega no lleva dicho bit activo, la función devolverá error ya que no se ha terminado de completar el proceso de establecimiento.
6. Estado ESTABLISHED: En este estado la conexión está establecida. Pueden suceder varios sucesos en función del paquete que entre:
   - Si lleva el bit FIN de las banderas activado: el host remoto quiere cerrar la conexión. Tras actualizar los números de acuse de recibo esperado y número de secuencia que se espera recibir se pasará al estado CLOSE_WAIT donde permanecerá hasta que se pretenda cerrar la conexión desde este mismo extremo. Por último se llamará a *mti_enviar_finack( )* para que se envíe un paquete con los bits FIN y ACK, del campo banderas, activos como respuesta al paquete entrante.
   - Si lleva simplemente la bandera ACK activa, es un paquete de asentimiento y se llamará a la función *mti_fin_procesoestados( )* para seguir con el proceso.
   - Si el campo banderas llevara activos otros bits que no fueran los que anteriormente se comentaron, la función devolvería error ya que, en este estado, no se puede dar otro caso distinto a las anteriores.
7. Estado CLOSE_WAIT: En este estado tendrá que permanecer hasta que no estén todos los datos enviados y se haya producido una llamada para cerrar la conexión. Por ello lo único que se hará será llamar a *mti_fin_procesoestados( )* que se encargará de enviar la trama con los datos que hubiera pendientes.
8. Estado LAST_ACK: A este estado se a pasado cuando ya se han enviado todos los datos pendientes y también se envió el paquete de FIN. Estará a la espera de que le llegue el paquete de contestación a éste, es decir, un paquete de FINACK. Si este es el paquete que ha llegado se pasará al estado CLOSED y se borrará la estructura que contenía toda la

información de esta conexión ya que, a partir de estos momentos, no existe (*mti_borrar_estructura(mti_conex)*).

9. Estado FIN_WAIT_1: La aplicación ha cerrado la conexión, pero el puerto remoto aún no. dependiendo del paquete que llegue, puede suceder:

   - Si el otro extremo de la conexión ha enviado un paquete de cierre de conexión (con el bit FIN activo), se pasará al estado CLOSING y se enviará un paquete de ACK (*mti_enviar_ack( )*).

   - Si el otro extremo de la conexión envía el asentimiento del paquete FIN, es decir, un paquete de FINACK, se pasará al estado FIN_WAIT_2 y permanecerá en este estado hasta que el host remoto también cierre la conexión. enviará un paquete de ACK.

   - Si el paquete de entrada sólo lleva activo el bit ACK, la conexión permanecerá en el mismo estado.

10. Estado FIN_WAIT_2: Si llega un paquete de FIN, se pasará al estado TIME_WAIT y se enviará como respuesta un paquete de FINACK. Si lo que ha llegado es un paquete de asentimiento se permanecerá en el mismo estado.
11. Estado TIME_WAIT: En este estado permanecerá un determinado tiempo para asegurar que ha llegado bien la trama de finalización de la conexión al otro extremo. En primer lugar se inicializará el temporizador2 a cero y permanecerá en un bucle mientras no se llegue a un determinado valor (se a programado para que sea 0.125 segundos ya que es un tiempo prudencial para que la trama llegue al otro extremo). Transcurrido este tiempo el estado de la conexión pasará a CLOSED y se borrará la estructura que mantenía la información de la conexión ya que la conexión ya está definitivamente cerrada.
12. Estado CLOSING: Si llega un paquete con la bandera FIN activa se pasará al estado TIME_WAIT. Si llega un paquete con datos, se contestará con un asentimiento.

- ***void mti_fin_procesoestados( )***: Función encargada de enviar datos, si los hubiera. Para entender mejor su funcionamiento se procederá a estudiar los casos posibles existentes:

1. Se busca si existen datos pendientes para enviar al host remoto *(mti_buscar_datoenviar(mti_conex->num_conex)*) y devolverá un puntero a dichos datos.

   - Si existen datos para enviar, se actualizarán en el buffer de salida (MTI_BUF->mti_datos[j]=punt_dato->info[j]), se activará la bandera ACK y se llamará a *mti_enviar_trama( )* para que rellene los campos de la cabecera y envíe la trama.

   - Si no existen datos para enviar, habrá que comprobar si se había producido anteriormente una llamada para cerrar la conexión y en qué estado se encuentra:

- ✓ Si el campo de la estructura que mantiene la información de la conexión está activo (mti_conex->enviar_fin == mti_si), será porque anteriormente se ha producido una llamada para cerrar el sistema. Si es así existen dos posibilidades:
  - Si el estado de la conexión era CLOSE_WAIT evolucionará a LAST_ACK y se enviará un paquete de FIN (*mti_enviar_fin( )*).
  - Si no estaba en ese estado, evolucionará a FIN_WAIT_1 y enviará un paquete de FIN (*mti_enviar_fin( )*).
- ✓ Si la conexión se encuentra en el estado CLOSE_WAIT es porque el host remoto envió un paquete para cerrar la conexión pero el host local aún no la ha cerrado, por ello tendrá que permanecer en este estado hasta que se produzca el cierre. Para informar de esta situación, se activa el campo mti_conex->espera_close=mti_si de la estructura de información.

- ***int mti_reset( )***: Cuando una conexión es abortada, es necesario informar al host remoto de lo sucedido y para ello habrá que enviar un paquete de reset.

  Para formar un paquete de este tipo, habrá que seguir los siguientes pasos:

  1. Se comprueba si el paquete que ha llegado ya es un paquete de reset y si es así, la función devolverá un error (-1) indicando que no se ha podido enviar dicho paquete. Ello es debido por que no se puede contestar a un paquete de reset con otro del mismo tipo.
  2. Si el paquete que ha llegado no era de reset, que tendrá que crear el paquete deseado. Para ello se tendrán que rellenar todos los campos de la cabecera:
     - El campo banderas se actualiza activando los bits ACK, asiente el paquete de entrada, y el bit RST, indica que el paquete que se envía es un reset.
     - El tamaño del dato será cero ya que no se podrán enviar datos en un paquete de este tipo.
     - El campo tcpoffset se actualizará con un valor de 5<<4. Informa de la longitud de la cabecera TCP expresada en palabras de 2 bytes. Se desplaza 4 bits debido a que el tamaño de este campo es 1 byte, donde los cuatro primeros bits son la longitud y los cuatro restantes están reservados.
     - El número de secuencia que se envía será igual al número de acuse de recibo del paquete que ha llegado (MTI_BUF->seqno[] = MTI_BUF->ackno[]).

- El número de acuse de recibo enviado será igual al número de secuencia que ha llegado incrementado en uno (MTI_BUF->seqno[] = MTI_BUF->ackno[]).

- El número de puerto fuente que se envía será igual al número destino que ha llegado en el paquete de entrada.

- El número de puerto destino que se envía será igual al número de puerto fuente que ha llegado en el paquete de entrada.

- La dirección IP fuente que se enviará en el paquete de reset deberá ser igual a la dirección IP destino del paquete entrante.

- La dirección IP destino del paquete que se enviará se actualizará con la dirección IP fuente del paquete de entrada.

3. Por último se llamará la función *mti_enviar_cabecera()* par actualizar el resto de los campos y enviar la trama creada.
4. La función devuelve 1 porque todo se ha realizado correctamente.

4.3.3 FUNCIONES AUXILIARES

4.3.3.1 Funciones relacionadas con el envío o la recepción de una trama

- ***void mti_guardar_trama(int i)***: Cuando llega una trama y se comprueba que la dirección IP es correcta pero que no es para el puerto que tiene abierto esa conexión, se debe de almacenar para que otra la pueda utilizar. Se almacenará en el array mti_BUFFER_PAQUETES y el elemento dentro de este array será el indicado por el valor que se le pasa como argumento. Como inicialmente MTI_BUF apunta a la trama que acaba de llegar todo su contenido será el que se tenga que almacenar en el array. Por ejemplo, se van a actualizar los cuatro primeros campos:

  - mti_BUFFER_PAQUETES[i].vhl=MTI_BUF->vhl;

  - mti_BUFFER_PAQUETES[i].tipo=MTI_BUF->tipo;

  - mti_BUFFER_PAQUETES[i].tam[0]=MTI_BUF->tam[0];

  - mti_BUFFER_PAQUETES[i].tam[1]=MTI_BUF->tam[1];

  Donde i representa el elemento dentro del array.

- ***void mti_trama_anter( )***: Su única función es la de almacenar en la estructura mti_trama_anterior la trama que se ha enviado. Ello es necesario ya que pueden existir problemas en la red y se tenga que volver a retransmitir la última trama que se envió.

- ***void mti_convierte_formatoSCI(INT8U *datos_SCI)***: Crea un array de octetos el cual estará formado por las cabeceras IP y TCP. Esto es necesario ya que para poder mandar datos por el puerto serie asíncrono(llamar a *enviar_SCI1(Baud_115200,cad,41))* sin problemas, es necesario que la cadena que se le pase esté formada únicamente por octetos y el buffer de salida con el que se está trabajando está formado tanto por bytes como por word. Se le parará la cadena datos_SCI por referencia y así todas las modificaciones que se realicen se verán reflejadas en la función que llame a ésta.

  El procedimiento que se debería seguir será:

  1. Llenar byte a byte todo el array con la información disponible. Por ejemplo, los cuatro primeros elementos serán:
     - datos_SCI[1]=MTI_BUF->vhl;
     - datos_SCI[2]=MTI_BUF->tipo;
     - datos_SCI[3]=MTI_BUF->tam[0];
     - datos_SCI[4]=MTI_BUF->tam[1];

- ***void mti_enviar_trama( )***: Actualiza algunos de los campos de la cabecera del buffer de salida MTI_BUF. ¿Por qué sólo actualiza algunos y la función *mti_enviar_cabecera()* actualiza el resto? Ello es debido a que, si se tiene que enviar un paquete de reset no hay que actualizar toda la cabecera, sino sólo una parte. Por ello la actualización de la trama se ha dividido en dos partes: esta función actualizará los campos que el paquete reset no necesita actualizar y llamará a *mti_enviar_cabecera()* para actualizar el resto de los campos.

  Los pasos a seguir son los siguientes:

  1. Actualizar los siguientes campos:
     - MTI_BUF->tcpoffset = 5 << 4. Informa de la longitud de la cabecera TCP expresada en palabras de 2 bytes. Se desplaza 4 bits porque el tamaño de este campo es 1 byte donde los cuatro primeros bits son la longitud y los cuatro restantes están reservados.
     - MTI_BUF->ackno[] = mti_conex->rcv_nxt[]. El campo número de acuse de recibo se actualizará al número de acuse de recibo esperado almacenado en la estructura.
     - MTI_BUF->seqno[] = mti_conex->snd_nxt[]. El número de secuencia se actualiza al número se secuencia esperado.
     - La dirección del puerto fuente que se envía será igual a la dirección local de la conexión y la del puerto destino a la del puerto remoto al

que se quiere enviar la información y que está almacenada en la estructura.

- La dirección IP fuente que se envía en la trama será igual a la dirección IP local de la conexión y la dirección IP destino será igual a la dirección IP del host remoto de la conexión.

2. Llamar a la función *mti_enviar_cabecera( )* para actualizar el resto de los campos.

- ***void mti_enviar_cabecera( )***: Rellenará los campos que no han sido actualizados en la función *mti_enviar_trama*.

  Los pasos a seguir serán:

  1. Actualizar los campos restantes. Los más significativos que se actualizan son:
     - Versión y longitud de la cabecera será 0x45.
     - El protocolo de capa superior será TCP por tanto el campo proto se actualiza IP_PROTO_TCP.
     - El campo tamaño se actualizará al tamaño total de la trama, es decir, a 40 que es el tamaño de la cabecera TCP e IP más el tamaño de los datos a enviar, si los hubiera.
     - Para calcular el checksum de la cabecera IP, primero se pone este campo a cero y después se llama a la función mti_ipchecksum() que nos dará el valor buscado.
     - Para calcular el checksum de la cabecera TCP y de los datos ser procederá igual que para calcular el checksum IP pero llamando a la función mti_tcpcheckusm().
  2. Llamar a la función *mti_convierte_formatoSCI()* para tener una trama que se pueda enviar por el puerto serie.
  3. Una vez que ya está la trama preparada se llamará a *enviar_SCI1()* para enviar la trama.

- ***void mti_enviar_ack( )***: Es llamada cuando se quiere enviar un paquete sin datos, sólo con un asentimiento. Para ello:

  1. Activa el bit ACK del campo banderas.
  2. Pone a cero la variable mti_tam_dato porque no se quieren enviar datos.
  3. Se llama a mti_enviar_trama() para actualizar todos los campos de la trama y enviarla.

- ***void mti_enviar_fin( )***: Si sólo se quiere enviar un paquete de fin se llamará a esta función. No se enviará ningún dato. El proceso es parecido al anterior:

  1. Activa el bit FIN del campo banderas.
  2. Pone a cero la variable mti_tam_dato porque no se quieren enviar datos
  4. Se llama a mti_enviar_trama() para actualizar todos los campos de la trama y enviarla.

- ***void mti_enviar_finack( )***: Enviará un paquete con los bits ACK y FIN activos para asentir un paquete de fin. No se enviarán datos. El proceso vuelve a ser similar al anterior:

  3. Activa los bits FIN y ACK del campo banderas.
  4. Pone a cero la variable mti_tam_dato porque no se quieren enviar datos
  5. Se llama a mti_enviar_trama() para actualizar todos los campos de la trama y enviarla.

4.3.3.2 Funciones que trabajan con listas enlazadas

- ***MTI_ESTRUCTURA_CONEX*mti_buscar_tcpestadobanderas(MTI_ESTRUCTURA_CONEX *conexion, int x)***: Recorrerá la lista donde se encuentran almacenadas todas las conexiones activas y buscará una de ellas que tenga el campo tcpestadobanderas igual al parámetro que se le pasa. También se le pasará como argumento el puntero a la estructura desde donde se debe empezar a recorrer la lista (no tiene porqué ser la primera). Si encuentra la estructura buscada devolverá un puntero a ésta. Si la búsqueda resulta negativa devolverá un puntero a NULL.

- ***int mti_buscar_conex_libre(void)***: Busca una conexión que no esté siendo utilizada, que esté libre. Para ello recorrerá el array mti_num_conexión donde se almacena para todas las conexiones si están libres u ocupadas. Devolverá el número de la primera conexión que encuentre que esté libre, actualizando ese elemento a ocupado ya que esa estructura va a ser utilizada. Si no encuentra ninguna el resultado será –1.

- ***MTI_ESTRUCTURA_CONEX * mti_buscar_id_conex(int x)***: Busca en la lista de conexiones activas aquella que tenga como identificador o número de conexión el valor que se le pasa como parámetro. Devolverá un puntero a la estructura encontrada, si es que ha hay, o puntero a NULL si no la encuentra.

- ***MTI_ESTRUCTURA_CONEX * mti_buscar_puerto(INT16U x)***: Recorre la lista de conexiones buscando una de ellas cuyo puerto local sea igual al que se

le pasa como argumento (INT16U x). Devolverá un puntero a la estructura que contenga dicho puerto (si existe) o a NULL si no lo encuentra.

- ***void mti_borrar_estructura(MTI_ESTRUCTURA_CONEX *conex)***: Borrará la estructura a la que apunta el puntero que se le pasa.

  Los pasos a seguir son:

  1. En el array mti_num_conexión, que mantiene información acerca de si una conexión está libre u ocupada, se actualizará a libre el elemento que informa acerca del estado de la estructura a borrar.
  2. Si la estructura a borrar es la primera y la única, se procederá a:
     - Apuntar mti_conexion a NULL.
     - Se liberará la memoria utilizada con *mti_free_conex()*.
  3. Si es la primera pero no la única:
     - mti_conexion apuntará al elemento siguiente al que se quiere borrar.
     - Se liberará la memoria utilizada con *mti_free_conex()*.
  4. Si está en medio de la lista o es la última:
     - Existirá un puntero auxiliar que apuntará al elemento anterior al que se pretende borrar para que, una vez borrado éste, la lista no se corte y el elemento anterior al borrado apunte al siguiente del mismo o a NULL si fuera el último elemento de la lista.

- ***SEND * mti_buscar_datoenviar(int x)***: Busca el primer dato almacenado cuyo identificar de conexión sea igual al que se le pasa como parámetro a esta función. Para buscar la estructura que contiene el dato buscado, se recorrerá la lista dinámica donde se almacenan los datos a enviar comprobando si su identificador coincide con el identificador de conexión *x*. Devolverá un puntero a la estructura que contiene el dato o NULL si no lo encuentra.

- ***void mti_borrar_datoenviado(int x)***: Borra la primera estructura que encuentre dentro de la lista de datos pendientes de envío cuyo identificador de conexión sea *x*(valor que se pasa como argumento).

  Los pasos a seguir son los siguientes:

  1. Si la estructura a borrar es la primera y la única, se procederá a:

- Apuntar mti_dato_a_enviar a NULL.
- Se liberará la memoria utilizada con *mti_free_enviar()*.

2. Si es la primera pero no la única:
   - mti_dato_a_enviar apuntará al elemento siguiente al que se quiere borrar.
   - Se liberará la memoria utilizada con *mti_free_enviar()*.
3. Si está en medio de la lista o es la última:
   - Existirá un puntero auxiliar que apuntará al elemento anterior al que se pretende borrar para que la lista no se corte y así, el elemento anterior al borrado apunte al siguiente del mismo o a NULL si fuera el último elemento de la lista.

4.3.3.3 Otras funciones

- ***INT16U mti_checksum(INT8U *datos, INT16U tam)***: Calcula el checksum de los datos que se le pasan como parámetro. También se le pasará el tamaño de los datos, expresados en word (2 bytes) no en bytes.

  El cálculo del checksum se realiza de la siguiente forma:

  1. Se suman todos los datos formando palabras de dos bytes.
  2. Cuando se han sumado todas las palabras, si existiera acarreo debido a que el tamaño de la suma es mayor de 2 octetos, éste se sumaría al resultado quedando así una palabra de tamaño word. Por ejemplo, si el resultado fuera 0x12a9b el uno sería acarreo y se sumaría a la b quedando 0x2a9c.
  3. Cuando ya se tiene el resultado se le calcula el complemento a uno y este será el valor del checksum. La función devolverá este valor.

- ***INT16U mti_ipchksum( )***: Calcula el checksum de la cabecera IP. Sólo llama a mti*_checksum()* pasándole como argumentos el puntero a la cabecera IP y el tamaño de ésta medido en word que, como no existe el campo de opciones, será diez. La función llamada devolverá el valor del checksum que, a su vez será el valor que devolverá *mti_ipchecksum()*.

- ***INT16U mti_tcpchksum( )***: Calcula el checksum de la cabecera TCP más los datos más una pseudocabecera formada por:

  - Dirección IP fuente.

- Dirección IP destino.
- 1 byte a cero.
- Tipo de protocolo del siguiente nivel.
- 1 byte a cero.
- Tamaño de la cabecera TCP más los datos.

El procedimiento será el siguiente:

1. Crear un array donde almacena la pseudocabecera actualizada con todos sus campos. Almacenará también la cabecera TCP y los datos. Si los datos no fueran múltiplos de 2 octetos, se rellenaría el último octeto con ceros.
2. Se llamará a la función *checksum* pasándole como parámetros:
   - El array que almacena toda la información
   - El tamaño. Para calcular éste, primero se calcula la suma de la cabecera TCP más la pseudocabecera que será 32. A esto se suma el tamaño del dato, si no es múltiplo de 2 se le suma 1 para que la suma sea divisible por 2(se rellena con ceros). El resultado será la suma en bytes. Como lo que se busca es el tamaño en word (2 octetos) se dividirá entre dos y ya se tiene en tamaño deseado para poder pasárselo a *mti_checksum()*.
3. Devolverá el resultado que le dé la función *mti_checksum()*.

- ***MTI_ESTRUCTURA_CONEX * mti_malloc_conex( )***: Buscará una estructura libre para poder almacenar la información de una nueva conexión. Para saber si una estructura está libre o no habrá que observar a qué valor está el campo num_conex de la estructura.

  - Si es –1 la estructura no está siendo utilizado y se podrá usar en la nueva conexión.
  - Si es un número distinto de –1 es porque esa estructura está siendo utilizada.

  En el proceso de inicialización del sistema o cuando se libera la estructura este campo se actualiza a –1 para que una conexión pueda utilizar dicha estructura.

  El procedimiento que seguirá esta función es:

  1. Ver si mti_conexion1 se puede utilizar, o lo que es lo mismo, ver si mti_conexion1.num_conex = -1. Si esto es cierto, ésta será la estructura que se busca y la función devolverá un puntero a ella. Si no es así se pasaría al siguiente paso.

2. Se realizaría la misma comprobación que se hizo con mti_conexion1 pero con mti_conexion2. Si se comprueba que no está siendo utilizada se devolvería un puntero a esta estructura y si no, se seguiría el proceso.
3. Se continuaría el proceso con las siguientes estructuras. Si se llega hasta mti_conexion5 y ésta también está ocupada el puntero que se devuelve apuntará a NULL ya que no se ha podido encontrar una conexión libre que poder utilizar.

- ***void mti_free_conex(MTI_ESTRUCTURA_CONEX *conexion)***: Libera una estructura que mantiene información de una determinada conexión para que otra la pueda utilizar. Se le pasará como argumento el puntero a la conexión que se quiere liberar. El único paso que tiene que realizar es actualizar el número de conexión a –1 (conexion->num_conex = -1) para indicar que esa estructura ya no está siendo utilizada y otra conexión la pueda utilizar.

- ***SEND * mti_malloc_enviar()***: Buscará una estructura libre para poder almacenar nuevos datos. El funcionamiento es el mismo que en la función *mti_malloc_conex*, así pues, para saber si una estructura se pude utilizar para almacenar datos o no, habrá que observar a qué valor está el campo num_conex de la estructura.

  - Si es –1 la estructura no está siendo utilizado y se podrá usar para almacenar datos.
  - Si es un número distinto de –1 es porque esa estructura está siendo utilizada.

  En el proceso de inicialización del sistema o cuando se borran datos este campo se actualiza a –1 para que otros datos a enviar puedan utilizar dicha estructura.

  El procedimiento que seguirá esta función es:

  1. Ver si mti_datos_a_enviar1 se puede utilizar (ver si mti_datos_a_enviar1.num_conex = -1). Si esto es cierto, ésta será la estructura que se busca y la función devolverá un puntero a ella. Si no es así se pasaría al siguiente paso.
  2. Se realizaría la misma comprobación anterior pero con mti_datos_a_enviar2. Si se comprueba que no está siendo utilizada se devolvería un puntero a esta estructura y si no, se seguiría el proceso.
  3. Se continuaría el proceso con las siguientes estructuras. Si se llega hasta mti_datos_a_enviar5 y ésta también está ocupada el puntero que se devuelve apuntará a NULL ya que no se ha podido encontrar una conexión libre que poder utilizar.

- ***void mti_free_enviar(SEND *datos)***: Libera una estructura de datos a enviar para que se puedan almacenar otros datos. Se le pasará como argumento el puntero a la estructura que se quiere liberar. El único paso que tiene que

realizar es actualizar el número de conexión al que iban dirigidos los datos a un valor de –1 (datos->num_conex = -1) para indicar que esa estructura ya no está siendo utilizada y se pueden almacenar otros datos.

- ***void mti_actualizar_rcv_nxt(INT16U n)***: Actualiza el campo número de acuse de recibo de la conexión con la que se esté trabajando en ese momento. Se le sumará el número n que se le pasa como argumento a mti_conex->rcv_nxt[].

  Los pasos que seguirá serán:

  1. Como n es una palabra de 2 octetos de tamaño y el número de acuse de recibo está formado por elementos de 1 octeto de tamaño, primero se sumará el byte menos significativo del número n al elemento menos significativo del número de acuse de recibo.
  2. Se sumará el byte más significativo de n al segundo byte menos significativo del número de acuse de recibo.
  3. Los dos octetos restantes se actualizarían si existiera acarreo.

  La función no devuelve ningún valor ya que está trabajando sobre una variable global.

- ***void mti_actualizar_ack_nxt(INT16U n)***: Actualiza el campo número de secuencia que se espera recibir. Se le sumará el número n que se le pasa como argumento a mti_conex->ack_nxt[].

  Los pasos que seguirá serán los mismos que en la función anterior.

4.3.3.4 Funciones para configurar los periféricos del microcontrolador

Estas funciones ya han sido diseñadas, así que la única que se comentará la función que es llamada cuando salta el temporizador debido a que ésta sí ha sido modificada.

- ***void isr_PIT1_PIF (void)***: Esta función es llamada cuando salta el temporizador del microcontrolador. Se encargará de incrementar los temporizadores temporizador1 y temporizador2 de todas las conexiones activas y que son utilizados en distintos puntos del proceso:

  - temporizador1: Se inicializa cuando se empieza a esperar una trama. Si transcurrido un determinado valor de éste aún no ha llegado la trama completa, se procederá a reenviar la anterior.

  - Temporizador2: Se inicializa cuando se llega al estado TIME_WAIT. Su única función es la de esperar un determinado tiempo antes de pasar al estado CLOSED.

5 PROTOCOLO DE PRUEBAS, EJEMPLOS Y APLICACIONES

Para comprobar el perfecto funcionamiento del proyecto así como para entenderlo un poco mejor, es necesario un protocolo de pruebas y algún ejemplo.

5.1 PROTOCOLO DE PRUEBAS

Durante el periodo de desarrollo de cada una de las funciones que componen la librería necesaria para implementar la pila TCP/IP, ha sido necesario un protocolo de pruebas para comprobar que todo funcionaba según lo previsto. Como era de esperar, durante el desarrollo de estas pruebas han surgido anomalías que se han tenido que corregir.

Durante este proceso, los pasos que se han tenido que seguir son los siguientes:

1. Inicialmente, todas las funciones han sido diseñadas y probadas en un PC con el programa "Borland C++ 4.52" corrigiendo los errores que iban surgiendo.
2. En paralelo a este proceso, se fue comprobado el funcionamiento de los periféricos del microcontrolador como son los temporizadores y el puerto serie. Para ello, se ha comunicando la tarjeta con el PC a través del Hyper Terminal y directamente escribiendo en el puerto serie.
3. Antes de comunicar un PC con la tarjeta, se comunicarán dos PC ya que de esta forma es más fácil depurar los posibles errores.
4. Por último, se ha comprobado el funcionamiento completo de todas las funciones en el microcontrolador utilizando para ello directamente el puerto serie del PC.

5.1.1 PRUEBAS REALIZADAS EN EL PC

Cada una de las funciones necesarias para implementar la pila TCP/IP se han diseñado en el lenguaje de programación C ya que es bastante parecido al lenguaje que utiliza el microcontrolador. Para ello se ha utilizado el programa "Borland C++ 4.52".

A continuación se van a describir los pasos seguidos para probar las funciones y los resultados que se han producido.

- Calculo del checksum o suma de verificación:

  A la primera función diseñada par calcular el checksum se le pasaba como argumento una cadena formada por elementos de 2 octetos de tamaño ya que este es el tamaño necesario para calcular la suma de verificación. Tras hacer varias pruebas con cadenas distintas y tras algunos arreglos, el funcionamiento fue el correcto ya que el resultado obtenido mediante esta función era igual al calculado manualmente.

Una vez realizado el cálculo correctamente hay que adaptar la función debido a que los datos de entrada no vienen expresados en palabras de 2 bytes de tamaño, sino en palabras de un octeto. Se comprobará que el resultado es correcto haciendo el cálculo manualmente.
Ya se tiene una función que calcula el checksum. Ahora habrá que adaptarla para que calcule la suma de verificación tanto de IP como de TCP.
Para IP es fácil, ya que el checksum sólo se calcula para la cabecera y por tanto sólo habrá que pasarle un puntero a ésta. Para TCP es un poco más complejo ya que habrá que añadirle una pseudocabecera. Los resultados comparados con los manuales resultan correctos.

- Comprobación de la cabecera:

  El siguiente paso es detectar una cabecera. Existe un paquete almacenado y se tendrá que comprobar si la versión es la correcta, si los checksum son correctos, si el nivel superior al que va destinado el paquete es TCP… También se podrá saber el estado de las banderas.

- Poner a la escucha un puerto:

  Para poner a la escucha un puerto simplemente habría que crear una estructura donde almacenar el número de puerto que se pone a la escucha, la dirección IP y el estado (LISTEN). El problema surge debido a que debe llevar asociado un identificador. Éste no debe representar a la conexión ya que aún no está abierta. Por ello se crea la variable mti_num_listen inicializada a un número mayor que el número máximo de conexiones posibles y que almacenará el último número que fue utilizado. Con ello se pretende que dos puertos a la escucha no tengan el mismo identificador. Para comprobar que funciona correctamente, se ponen varios puertos a la escucha y se manda a pantalla el identificador de cada uno. Se comprueba que son distintos entre ellos y distintos a los posibles números que pueden identificar las conexiones.

- Establecer una conexión:

  En la apertura de una conexión forman parte tanto el cliente como el servidor. Se estudiará como reacciona cada uno de ellos según el paquete que reciba durante el establecimiento.

  - Cliente: Para abrir una conexión por parte del cliente, el primer paquete que se debe enviar debe llevar el bit SYN activo. Ello se consigue llamando a la función mti_connect(). Para comprobar que el paquete que se enviará tiene actualizados todos los campos, se imprimirá por pantalla y se comprueba que el paquete se ha formado correctamente.

  - Servidor: El segundo paso para abrir la conexión es la aceptación por parte del servidor. Anteriormente a la aceptación se ha puerto el puerto a la escucha. Se llamará a la función mti_accept() pasándole como buffer de entrada el paquete de SYN que envió la función mti_connect(). El paquete que se forma como respuesta a éste será un paquete de SYNACK. Ello se comprueba mandándolo a pantalla y observando que era el esperado.

    Para chequear los posibles errores, se varía el checksum al paquete que se envía y se comprueba que la función mti_accept() devuelve error(–1) ya

que cuando compare el checksum que le llega con el que ella ha calculado comprobará que son distintos (se ha comprobado variando tanto el checksum IP como el TCP). Ello se comprueba observando que no crea ninguna trama de respuesta y mandando a pantalla el valor que retorna (se será –1).

Otro posible caso que podría suceder es que el paquete que llega no sea para el puerto que está a la escucha. En este caso la función devolverá 3 y no enviará ninguna trama de respuesta ya que el paquete no era para ella. Se comprobará mandando a pantalla el valor devuelto por la función y observando que es el que se esperaba según la situación.

Otro de los posibles errores es que la dirección IP sea distinta. La función devolverá 2 y no responderá con ninguna trama ya que, al igual que en el caso anterior, la trama no es para ella. La comprobación se vuelve a hacer enviando el valor devuelto a la pantalla.

Todos los posibles errores comentados anteriormente, no solo se pueden dar en el establecimiento de la conexión, sino que pueden suceder en cualquier trama que se transmita por la red. Por ello, aunque está comentado para la trama de apertura de conexión, también se realizaron pruebas en el intercambio de datos y el resultado fue el mismo.

- Cliente: Cuando al cliente envió la trama de conexión pasó al estado SYN_SENT. Por ello, la única trama que admitirá será una trama de SYNACK. En el buffer de entrada se coloca esta trama y se comprueba, enviando la respuesta a pantalla, que responde correctamente. Para comprobar el funcionamiento, se envió una trama en la que el bit SYN del campo banderas no estaba activo, y se comprobó que se producía un error.

- Escritura de datos:

  Cuando se quieren escribir datos para enviarlos al otro extremo, se llama a la función mti_write(). Ésta los almacena en una estructura de donde se tomarán cuando se puedan enviar al par.

  Existen varias posibles situaciones:

  - Si existe sitio en la lista para almacenar estos datos, se almacenarán. Ello se comprobará observando que el valor que devuelve es 1.
  - Si antes de hacer la llamada para escribir dato se hubiera mandado cerrar la conexión, no se podría escribir el dato, y ello se comprobará observando que el valor devuelto por la función es –1.
  - Otra posible situación es que no exista espacio en la lista para almacenar los datos. Como en el caso anterior se podrá comprobar mandando a pantalla el valor devuelto por la función y observando que es –1.

- Lectura de tramas:

  Cuando se desea leer una trama, puede que llegue bien o que existan errores. Los posibles casos que se han comprobado se detallan a continuación:

  - La trama ha llegado bien. La prueba de ello es que, mandando a pantalla la trama que envía, se observa que todos los campos están actualizados. Otra

posible comprobación es observar el valor que devuelve la función y comprobar que es 1.

- El caso en el que existe error en el checksum, en el número de puerto o en la dirección IP se comentaron anteriormente ya que el procedimiento para probarlo ha sido el mismo.
- Si el número de asentimiento llegara mal, la función deberá enviar el paquete anterior indicando que ha existido algún error. La comprobación es fácil, En el buffer de entrada se coloca la trama anterior a la que debería de ser y se comprueba que el host reenvía la trama anterior. También se ha comprobado que si, esto se repite más de tres veces, la trama no se volverá a enviar más.
- Otra de la prueba a realizar es observar el funcionamiento si no se envía ninguna trama y el host se queda a la espera. Se comprueba que, en este caso, transcurrido un determinado tiempo, el host vuelve a enviar la trama anterior. Si sigue sin enviársele ninguna trama volverá a reenviarla, hasta un máximo de tres veces.

- Abortar una conexión:

  Cuando se aborta una conexión, se crea una trama de reset para enviarla al otro extremo. Cuando se hace una llamada para abortar, se comprueba que se ha realizado correctamente observando que el paquete que envía es un reset y que todos sus campos son correctos. Esto sucede si el abortar la conexión se ha producido correctamente ya que pueden existir varios casos en los que produciría error.
  - Si se intenta abortar una conexión que no existe, la función devolverá -1 y no enviará ningún paquete. Ello se puede comprobar llamando a la función mti_abort() con un número de identificador distinto al de las conexiones activas en ese momento.
  - Si la conexión ya se estuviera cerrando, pasaría lo mismo. La comprobación es sencilla, primero se llama a mti_close() y posterior mente a mti_abort(). Se comprueba que la segunda función devuelve –1.

- Cerrar una conexión:

  La comprobación del cierre de la conexión se comprueba observando si el paquete que se crea es un paquete de FIN y el paquete que le contesta es uno de FINACK. También hay que tener en cuenta que todos los campos sean correctos. Al igual que en el caso anterior, pueden existir errores:
  - Puede suceder que la conexión no exista y la función devolverá –1. Ello se comprueba intentando cerrar una conexión cuyo identificador sea distinto de las conexiones activas. La función devolverá un valor que se podrá enviar a pantalla y comprobar que es –1.
  - También pude suceder que la conexión ya esté en proceso de cerrarse. Para ver lo que pasaría, se llamará dos veces seguidas a la función mti_close() con el mismo identificador. La primera devolverá 1 ya que sí puede cerrar

la conexión y la segunda devolverá –1 pues detectará que ya está cerrándose.

5.1.2 PRUEBAS REALIZADAS CON LOS PERIFÉRICOS DEL MICROCONTROLADOR

Para comprobar el funcionamiento de los periféricos del microcontrolador, se van a hacer pequeños programas para ver su modo de funcionamiento. Las funciones utilizadas ya estaban realizadas y sólo ha sido necesario estudiar su funcionamiento y los parámetros que necesitan.

- Puerto serie asíncrono: Hay que comprobar que el puerto transmite y recibe. Para ello se abre una comunicación entre el micro y el PC. La comunicación pude realizarse de dos formas, ya sea utilizando el Hyper Terminal o escribiendo directamente sobre el puerto serie del PC.
  - Utilizando el Hyper Terminal sólo habrá que configurarlo para que su velocidad sea la adecuada al microcontrolador y sin paridad.

    Para transmitir se utiliza una función ya diseñada a la que hay que pasarle como parámetro la velocidad de transmisión, que será igual a la configurada en el PC. También se le pasa un puntero a los datos que se quieren transmitir y loa longitud de estos.
    La recepción será por interrupciones y existirá una función ya diseñada encargada de almacenar cada vez que llegue un dato a la entrada en un buffer.
    El primer programa diseñado realizaba un eco de la señal de entrada. La comprobación era fácil, se enviaba un carácter por el PC y el mismo carácter era recibido poco después.
    El segundo programa intentaba realizar algo más que un simple eco. Esperaba hasta que llegara una trama de nueve octetos. Si el primero de ellos, considerado como una cabecera, correspondía con “0x61” o, lo que es lo mismo, una “a” en código ASCII que es el formato con el que trabaja el Hyper Terminal, la trama sería correcta y respondería enviando una trama determinada de tres bytes. Si la cabecera no fuera la deseada enviaría una trama distinta. Tras realizar varias pruebas enviando caracteres distintos, se comprueba que el funcionamiento es el deseado.
  - Para escribir directamente sobre el puerto serie del PC, habrá que hacer un programa en el que, inicialmente se configure el puerto a la velocidad que se desea, con un bit de start y otro de stop. También habrá que diseñar dos funciones para la lectura y escritura de datos. Para leer hay que esperar a que existan datos en el buffer de entrada y leerlos de un determinado registro. Para escribir hay que espera hasta que el buffer esté vacío y escribir en el registro.

    Para comprobar que se envían y se reciben datos, se realizan las mismas pruebas que en apartado anterior, donde ahora para escribir habrá que llamar a una función y para leer a otra. Aquí los datos se le envían directamente en hexadecimal.

- Temporizadores: Existe una función que puede ser programada para que cuente un determinado valor y, una vez que haya terminado, salte una interrupción. La rutina de interrupción se puede programar para que realice distintos eventos.

  La primera prueba realizada es encender y apagar un led cada vez que salte la interrupción debida al temporizador. Poniendo un contador no muy pequeño ni muy grande, se puede observar que el led se enciende y se apaga a la misma frecuencia, como era de esperar.
  Otra de las pruebas realizadas fue la siguiente. En el programa existe una variable que se irá incrementando en la rutina de interrupción. Habrá un bucle que irá comprobando el valor del contador, para que, llegado un valor determinado, se encienda un led. Se comprueba que el funcionamiento es correcto. Otra comprobación es variar un la función el valor que debe contar el temporizador. Si se aumenta este valor se observa que el led tarda más tiempo en encenderse al igual que si el valor es menor, tardará menos tiempo como era de esperar.

5.1.3 PRUEBAS REALIZADAS CON DOS PC

Antes de comprobar el funcionamiento con el microcontrolador, se comunicarán dos PCs y se intentará establecer una comunicación entre ellos.

Para realizar esta comunicación, habrá que escribir directamente sobre el puerto serie del PC. Para ello, inicialmente se configurarán los registros necesarios para que la comunicación sea la velocidad requerida y el tamaño del dato sea el deseado.

En primer lugar se establecerá una conexión entre los dos PCs o entre los dos puertos de un mismo PC. Para comprobar que el establecimiento de la conexión es correcto, se imprimirá por pantalla cada una de las tramas que se transmiten en este proceso, comprobando que son las necesarias, es decir, igual a las calculadas teóricamente.

Tras esta conexión, se procederá al flujo de datos. Se comprueba que aparecen problemas debido a los temporizadores y las retransmisiones que no aparecían al utilizar un solo PC. Tras solucionar este problema se procederá a probar el proyecto utilizando el microcontrolador.

5.1.4 PRUEBAS REALIZADAS CON EL PROGRAMA GENERAL EN EL MICROCONTROLADOR

Para realizar las comprobaciones con el programa completo, se establecerá una comunicación con el PC y la tarjeta hardware a través del puerto serie del PC. Se supondrá que, inicialmente, la tarjeta actúa de cliente y el PC de servidor. Para ello, se actuará de la siguiente manera:

- Inicialmente se intentará abrir una conexión. El PC se pondrá a la escucha y esperará a que le llegue una trama de apertura de conexión por parte de la

tarjeta. Para comprobar que el funcionamiento es el deseado, tanto las tramas que envía el cliente, como las que llegan desde el micro, se enviará a pantalla para poder ir depurando el programa. Otra comprobación que se realizará en la tarjeta, será encender un led si se produjera algún error. Con este paso se habrá establecido una comunicación.

- Otra de las pruebas realizadas es el cierre de la conexión tras haberla abierto. Como en el caso anterior, para comprobar el correcto funcionamiento, se enviarán las tramas recibidas y enviadas a la pantalla del PC para poder ir depurando.
- También habrá que comprobar qué sucede si se aborta la conexión por parte de la tarjeta. Para ello el micro se programará para que encienda el led si, al llamar a la función de abortar, todo se ha realizado correctamente. En el PC se comprobará si la trama que llega es un reset.
- Por último, se comprobará que sucede cuando se transmiten datos. Para ello, el micro intentará enviar una cadena de caracteres al PC tras haber abierto la conexión. Al ir enviando a pantalla las tramas, se comprueba que todos los campos de éstas son correctos, es decir, se comprueba que el número de secuencia y de asentimiento son los deseados, así como el checksum.

5.2 EJEMPLOS

Se realizarán varios ejemplos que complementarán al protocolo de pruebas para comprobar el correcto funcionamiento del programa.

- Realización de eco: La tarjeta realizará un eco de los caracteres que lleguen procedentes del PC.
  - Primero se abre una conexión.
  - Tras la apertura, se pedirá que se introduzca un dato por teclado.
  - Este dato será enviado al microcontrolador y este lo leerá y lo enviará de vuelta al PC.
  - En el PC se imprimirá por pantalla y se comprobará si es el mismo dato que se envió.

- Búsqueda de una clave: El microcontrolador actuará como servidor que tiene una clave oculta. El PC actuará como cliente que desea adivinar esa clave.
  - El PC pedirá que se introduzca un dato por teclado.
  - Este dato será enviado a la tarjeta y ésta, en función si el dato coincide con la clave que tiene almacenada, enviará un mensaje de “error” o de “ok”.
  - El cliente leerá el mensaje que le ha llegado y comprobará si la clave que envió es la correcta o no.

Se realizaran varias pruebas para comprobar que el resultado es le correcto. En cada una ade ellas se introducirán distintos valores comprobando que da error, hasta se se introduce el carácter c que es la clave y la función devolverá ok.

- Prueba 1:

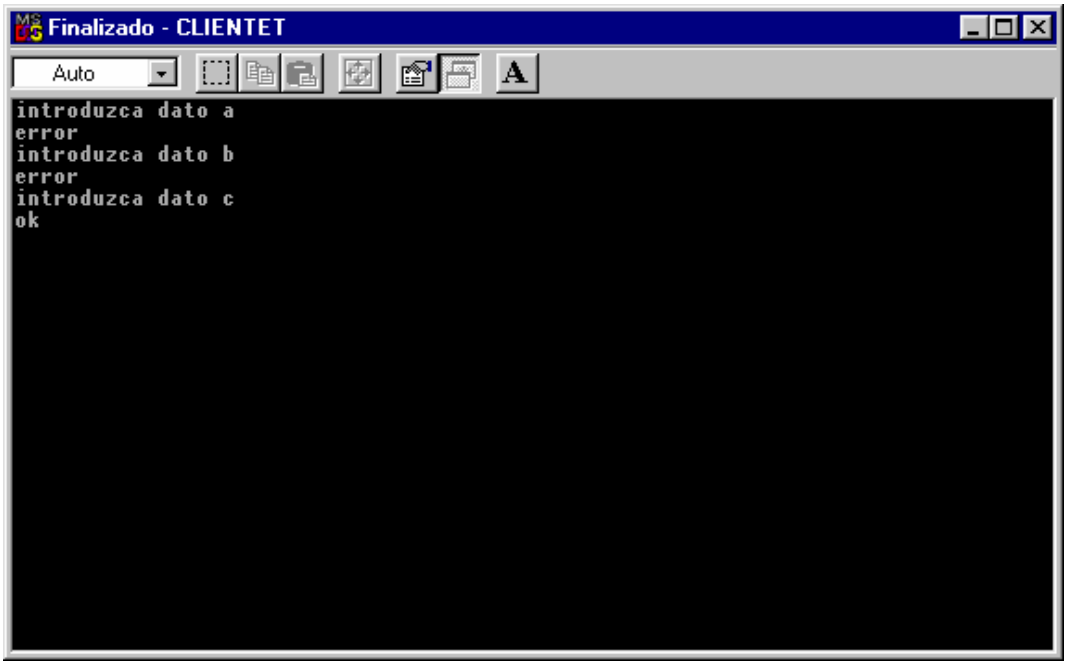

Figura 5.1

- Prueba 2:

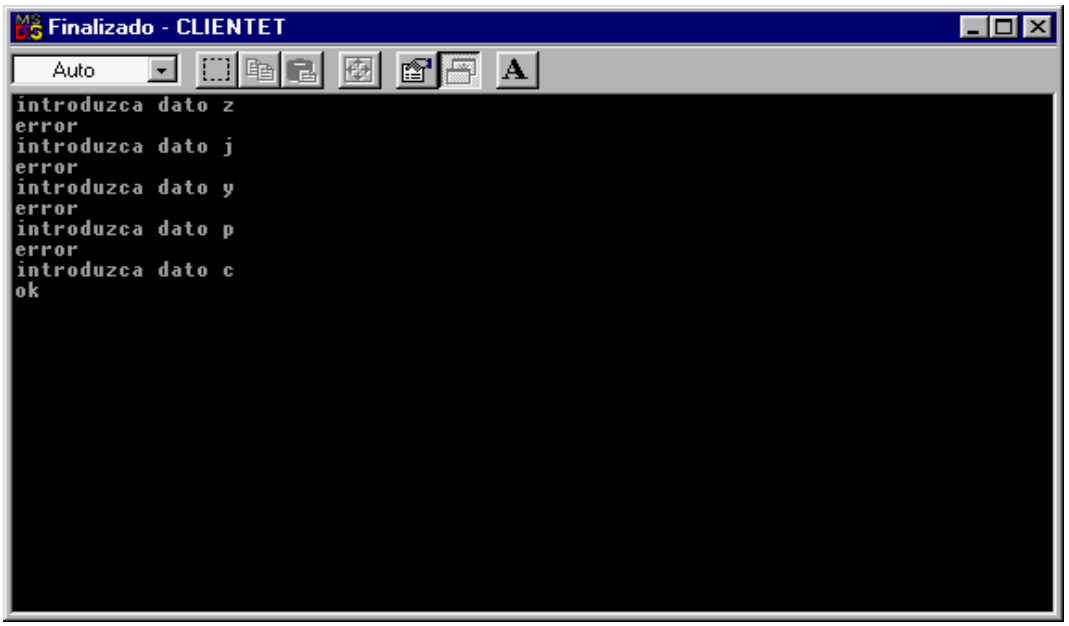

Figura 5.2

- Prueba 3:

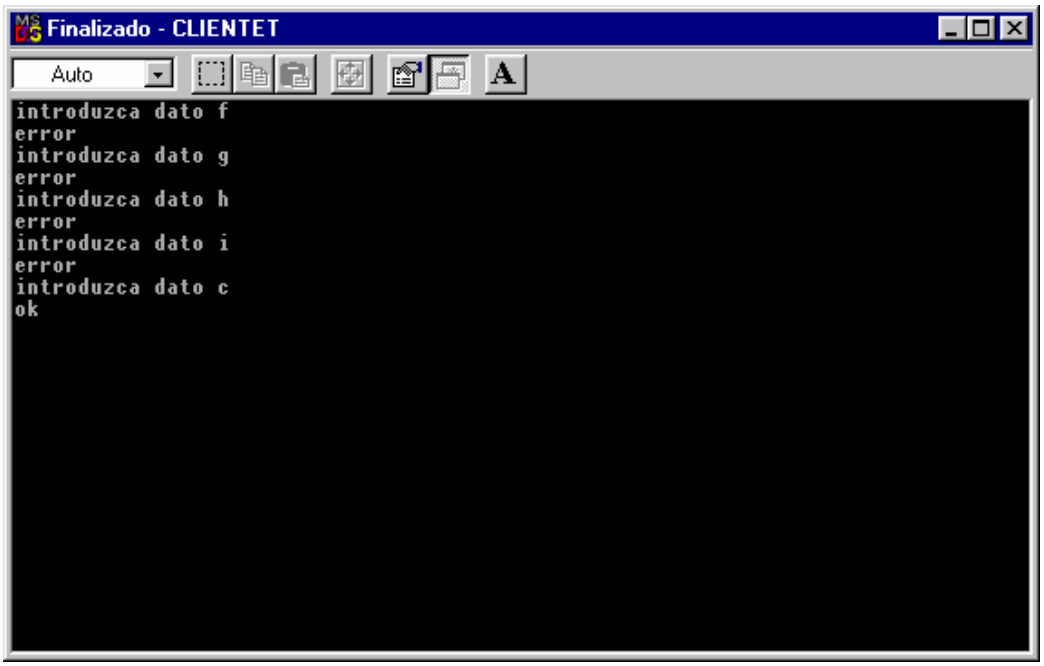

Figura 5.3

5.3 APLICACIONES

Existen diversas aplicaciones en las que se podría utilizar una pila TCP/IP y aprovechar las ventajas que nos ofrece frente a otros protocolos como son fiabilidad y seguridad. A continuación se van a comentar algunas de ellas:

- TELNET: Es una aplicación que permite desde un determinado lugar y con el teclado y la pantalla de un PC, conectarse a otro remoto a través de la red. Lo importante, es que la conexión puede establecerse tanto con una máquina multiusuario que está en la misma habitación o al otro lado del mundo. Una conexión mediante Telnet permite acceder a cualquiera de los servicios que la máquina remota ofrezca a sus terminales locales. De esta manera se puede abrir una sesión (entrar y ejecutar comandos) o acceder a otros servicios especiales: como por ejemplo consultar un catálogo de una biblioteca para buscar un libro, leer un periódico electrónico, buscar información sobre una persona, etc.
- FTP (Protocolo de Transferencia de Ficheros): Una de las operaciones que más se usa es la copia de ficheros de una máquina a otra, es decir, para transferencias de alta velocidad de un disco a otro. El cliente puede enviar un fichero al servidor. Puede también pedir un fichero de este servidor.
- SMTP (Simple Mail Transfer Protocol): Define el mecanismo para mover correo entre diferentes máquinas, es decir, como sistema de correo de Internet.

- Control de la posición de un heliostato: Existe un heliostato encargado de seguir el movimiento del sol para poder almacenar la mayor cantidad de energía posible. Este proceso se controla mediante un microcontrolador. Se podría utilizar una pila TCP/IP para tener en todo momento informado a un operario que no se encuentra en el lugar de la posición en que se encuentra el helióstato, al igual que el operario podría mandar ordenes para que se moviera si se quisiera que estuviera en otra posición.
- Control de temperatura en una fábrica: Mediante sensores se toman muestras de la temperatura que existe en cada parte de la fábrica. Mediante TCP/IP se mandará esta información a una sala de control que se encargará de subir o bajar la temperatura mandando la orden correspondiente, también mediante TCP/IP, al aparato de aire acondicionado.
- Intercambio de información en un hospital: Se puede utilizar TCP/IP para mandar información acerca de las constantes vitales y estado de un paciente a una sala de enfermeras.

6 LIMITACIONES Y POSIBLES AMPLIACIONES

En este proyecto se han desarrollado todas las funciones necesarias para realizar una pila TCP/IP para una posible comunicación. Por diversos motivos, dicho proyecto tiene unas determinadas limitaciones que podrían subsanarse en futuras ampliaciones.

6.1 LIMITACIONES

La primera limitación que aparece es el número máximo de conexiones que se pueden mantener activas. Se ha considerado en este proyecto que sea cinco. La razón por la que se ha decidido no tener más de estas conexiones es porque cuando se abre una conexión, se crea una estructura que mantendrá toda la información necesaria. Esto ocupa memoria y si existieran muchas conexiones se utilizaría mucha memoria y el rendimiento del programa disminuiría.

La segunda limitación es el número máximo de veces que se puede retransmitir un paquete. Cuando no llega la respuesta a un paquete enviado, éste se debe retransmitir, pero si ello sucede varias veces puede ser debido a que exista un problema en la red o en el otro extremo y el paquete nunca llegue. Por ello si no existiera un número máximo de retransmisiones, se podría transmitir indefinidamente. En el proyecto se ha tomado este número como tres, aunque cualquier usuario que utilice estas funciones lo puede variar y poner un número mayor.

El número de datos almacenados esperando ser enviados también está limitado. Sucede lo mismo que con las conexiones, cuando se almacena un dato se crea una estructura que ocupa memoria. Por ello, el número máximo de datos que pueden existir almacenados está limitado.

En la cabecera tanto de TCP como de IP, existen unos campos que permiten que se pueda realizar fragmentación de paquetes. Incluso añadir opciones. En este proyecto no existirá ni la fragmentación, ni las opciones.

6.2 POSIBLES AMPLIACIONES

Las posibles ampliaciones que se pueden realizar en este proyecto son varias. La primera de ellas podría ser mejorar todas las limitaciones que tiene esta proyecto. Para ello, se podría añadir a la placa algún dispositivo de memoria para almacenar datos. Así se podrán tener más conexiones abiertas y más datos almacenados sin preocuparse por la memoria ocupada pos sus estructuras. También se podría ampliar configurando la cabecera para que pueda existir fragmentación y opciones.

Aquí se ha desarrollado los protocolos de las capas de transporte y de Internet de la pila TCP/IP. Una ampliación de este proyecto sería implementar también un protocolo de la capa de interfaz de red. Éste podría ser el protocolo PPP o Protocolo Punto a Punto.

7 BIBLIOGRAFÍA

- Protocolos de Internet. Diseño e implementación en sistemas UNIX. Ángel López y Alejandro Novo. De. Ra-Ma. Noviembre 1999.
- Redes globales de información con Internet y TCP/IP. D.E. Comer
- RFC 768: Protocolo de Datagramas de Usuario.
- RFC 791: Protocolo de Internet.
- RFC 793: Protocolo de Control de Transmisión.
- RFC 1549 y RFC 1661: Protocolo Punto a Punto.
- http://monografías.com/trabajos/protocolotcpip: resumen de TCP/IP, capas del modelo OSI y moelo estratificado por capas de Internet.
- http://ditec.um.es/laso/docs/tut-tcpip/3376fm.html: Tutorial y descripción técnica de TCP/IP.
- http://usuarios.lycos.es/janjo/janjo1.html: Funcionamiento de TCP/IP.
- www.ii.aum.es/~mruiz/redesll/ipcheck.htm: Informa de cómo calcular el checksum.
- http://gsyc.escet.urjc.es/docencia/asignaturas/redes/transpar/nono3.html: Descripción del protocolo PPP.
- http://isc.tipod.com.mx/ndis.htm: Normas de interconexión del puerto serie del PC.

8 ANEXOS

8.1 ANEXO 1: MANUAL DE USUARIO

El siguiente documento es un manual para que un usuario que pretenda enviar una información a la red usando una pila TCP/IP, pueda realizarlo sin problemas. Se necesita conocer qué funciones se deben utilizar en cada momento, los parámetros que necesitan cada una de ellas, así como los valores que devuelven.

Para hacer compatible el tipo de datos con los del programa, el usuario debe saber que:

- INT8U equivale a unsigned char: Palabra de ocho bits (un byte).
- INT16U corresponde con unsigned short: 2 bytes.

- **Paso 1: Inicialización del sistema.**

  Cuando un usuario quiere establecer una conexión lo primero que debe hacer es inicializar el sistema. Se deben inicializar los registros, punteros, variables, … También se tendrán que configurar los periféricos del microcontrolador que se van a utilizar, tanto el puerto serie asíncrono como el temporizador. Para realizar toda la inicialización el usuario deberá llamar a la función *mti_init()*:

  - ***void mti_init (INT8U local_addr[])***: Es la primera función que se llamará ya que es la encargada de inicializar el sistema de red. Para ello inicializará todos los punteros, variables necesarias en el proceso… Hay que pasarle como parámetro la dirección IP local. Ésta está formada por cuatro números enteros separado por puntos, por ello lo que se pasa como parámetro será una cadena de cuatro elementos donde cada uno de ellos será uno de estos números enteros. Por ejemplo:
    - Si una dirección IP es 193.147.1.1 lo que le pasamos a la función será local_addr[4] = {193,147,1,1}

- **Paso 2: Apertura de la conexión.**

  Una vez que el sistema ya está inicializado se debe abrir la conexión. Ésta se puede realizar de dos formas, según si el usuario quiere hacer las funciones de cliente (se abre directamente la conexión) o de servidor (primero se pone el puerto a la escucha y posteriormente se acepta la conexión):

  - ✓ Si se actúa como cliente, éste será el encargado de empezar el establecimiento de la conexión. Para ello tendrá que crear una trama en la que el bit SYN del campo banderas vaya activado para que el servidor pueda saber que se trata de un establecimiento de la conexión. La función que deberá ser llamada para este fin será *mti_connect()*:

- ***int mti_connect (INT8U rem_addr[], INT16U rm_puerto)***: Una vez que se ha inicializado el sistema, esta función enviará una solicitud de conexión. Internamente ésta se encargará de buscar un puerto local para utilizarlo, crear la estructura con la información necesaria para la conexión, crear el paquete que se enviará al par remoto y enviarlo. Todo esto queda fuera del interés del usuario. Él sólo tiene que saber los parámetros que se le tienen que pasar y el significado del resultado que devuelve.

  Los parámetros que se le pasan son:

  - rem_addr[]: Dirección IP del host remoto al que se quiere conectar. Al igual que en la función *mti_init()*, esta dirección vendrá dada como una cadena de cuatro elementos de tamaño INT8U.
  - rm_puerto: dirección del puerto remoto al que se quiere conectar. Será una palabra de 16 bits (INT16U).

  Los valores posibles que puede devolver dependen de si la solicitud conexión se ha realizado correctamente o de si ha existido algún error. Estos valores serán:

  - -1: la función retornará este valor cuando se ha producido algún error.
  - Número distinto de -1: si la solicitud se ha realizado correctamente la función devolverá el número que identificará de la conexión y será el valor que se le pasará al resto de las funciones cuando queramos enviar datos, leerlos, cerrar la conexión o abortarla.

✓ Si la conexión se realiza en dos pasos, primero habrá que poner a la escucha el puerto que se quiere conectar con el host remoto. Para ello se utiliza la función *mti_listen()*. Una vez que el puerto se encuentra a la escucha se procederá a aceptar la conexión. Para ello habrá que esperar una trama de apertura de conexión y si todo es correcto se responderá con otra trama de confirmación de conexión. Todo ello lo realiza la función *mti_accept()*.

- ***int mti_listen (INT8U local_addr[], INT16U local_port)***: Se encarga de poner un puerto a la escucha para esperar que le llegue un paquete de solicitud de conexión proveniente de un host remoto que quiere conectarse a éste. Los parámetros que necesita son:

  - local_addr[]: Dirección IP del host. Vendrá dada como una cadena de cuatro elementos tipo INT8U.
  - rm_puerto: Dirección del puerto local que ponemos a la escucha.. Será una palabra de 16 bits.

  La función retronará distintos valores según se haya realizado la puesta a la escucha. Éstos serán:

  - -1: la función retornará este valor cuando se ha producido algún error.

- Número distinto de -1: El puerto ha sido puesto a la escucha correctamente. Este número identificará el puerto a la escucha, no la conexión.

- int mti_accept(int x): Debe ser llamada cuando, tras tener un puerto a la escucha, se desea aceptar una solicitud de conexión para ese puerto. Para ello se espera hasta que llegue un paquete completo con una solicitud de conexión y, tras comprobar que el paquete es correcto (comprobación del checksum, de las banderas…) se enviará un paquete con la confirmación de la conexión. Si transcurrido un tiempo desde que se desea aceptar la conexión no llegara un paquete de apertura de conexión se rechazaría la conexión.

  El parámetro que se pasa será:

  - x: Número entero que identifica el puerto que está a la escucha y que se quiere conectar.

  La función devolverá diversos valores según haya podido o no procesar el paquete de solicitud de conexión y crear el de aceptación. Estos valores serán:

  - -1: la función retornará este valor cuando se ha producido algún error ya sea al procesar el paquete de entrada o al crear el nuevo.
  - Número distinto de -1: si la aceptación de la conexión se ha realizado correctamente, la función devolverá el número que identificará la conexión y será el valor que se le pasará al resto de las funciones cuando se quieran enviar datos, leerlos, cerrar la conexión o abortarla.

- **Paso 3: Datos para enviar.**

  Cuando el usuario desea enviar datos al par remoto, sólo necesitará llamar a la función *mti_write()* y esta se encargará de almacenar los datos para que puedan ser enviados.

  - ***int mti_write (int socketesclavo, INT8U * mensaje, INT16U tam_mensaje)***: Es la encarga de poner en una cola de envío la información que se quiere enviar al par. Cuando se forme el paquete para el par, se buscarán en esta lista datos pendientes de envío y se encontrarán los almacenados con la llamada a esta función.

    Los parámetros que le tienen que pasar serán los siguientes:

    - int socketesclavo: Número entero identificador de la conexión a la que se le quiere enviar el mensaje.
    - INT8U * mensaje: Puntero a la información que se desea enviar al host remoto. El mensaje debe estar formado por elementos de tipo INT8U.
    - INT16U tam_mensaje: tamaño del mensaje que se envía. Será de tipo INT16U.

La función devolverá distintos valores en función de si todo se ha realizado correctamente o no. Éstos son los siguientes:

- -1: Se ha producido un error al almacenar el dato en la lista o no se puede escribir porque anteriormente a la llamada a esta función se intentó cerrar la conexión.
- 1: Todo se ha realizado correctamente, el mensaje que se quiere enviar al par ha sido colocado correctamente en la cola.

- **Paso 4: Lectura de datos**.

Cuando se pretenda leer una trama y obtener de ellas datos (si es que transportara información), será necesario llamar a la función *mti_read()*. Una vez realizada esta llamada, los datos que transporte estarán almacenados en el array buffer, que se le pasa como parámetro, y de él se deberán extraer.

- ***int mti_read (int socket, INT8U *buffer, int tam_buffer):*** Lee los datos que llegan por el buffer de entrada. Espera hasta tener un paquete completo y lo procesa (comprueba si el dato es para la conexión activa, le calcula el checksum, comprueba si el asentimiento y el número de secuencia son correctos, en definitiva, comprueba si el dato es correcto). Una vez procesado, crea un paquete de respuesta y lo envía junto con los datos que estuvieran esperando para ser enviados a esta conexión (si los hubiera).

Los parámetros que se le pasan son los siguientes:

- int socket: Identificador de la conexión de la cuál se quiere leer el paquete de llegada y extraer la información. Será un número entero.
- INT8U *buffer: Puntero al buffer donde se almacenará, tras procesar la cabecera, la información, en bytes (INT8U), que es enviada por el host remoto. El usuario podrá procesarla ya que el objetivo de la conexión es el intercambio de información. En el caso de que el paquete no llevara datos (fuera un paquete de solicitud de conexión, de fin de conexión…) este buffer estaría vacío.
- int tam_buffer: Número entero que nos informa del tamaño de buffer anterior.

La función podrá devolver varios valores en función de haber realizado todo correctamente o no. Éstos son los siguientes:

- -1: Se ha producido un error al procesar el paquete de llegada o al crear el paquete respuesta.
- 1: Todo se ha realizado correctamente, el paquete de llegada es correcto y no ha habido problemas al crear el paquete.

- **Paso 5: Cierre de la conexión**

Una vez que ya no sea necesaria mantener una conexión abierta se procederá a su cierre. Éste se puede realizar de dos formas según se cierre correctamente o se aborte la conexión sin tener en cuenta los datos pendientes de ser enviados.

✓ Si la conexión se desea cerrar correctamente se llamará a *mti_close()*. Con esta llamada de hace saber que se desea cerrar, pero antes habrá que enviar toda la información que esté pendiente.

- ***int mti_close (int numero_de_conexion)***: En el momento en el que se desee cerrar la conexión, habrá que llamar a esta función. El cerrar la conexión no será algo brusco sino que habrá que esperar a que se envíen todos los datos que estén en la cola. También se aceptarán todos los datos que lleguen del par remoto y por último se esperará hasta que llegue el fin de cierre de la conexión por parte del host remoto. Una vez que se haya llamado a esta función, una llamada a la función *mti_write()* dará error ya que no se pueden enviar más datos, sólo se enviarán los que ya estaban en la cola antes de llamar a *mti_close()*.

  Sólo se le pasa como argumento un parámetro:

  - int numero_de_conexión: Número que identifica la conexión que se quiere cerrar. Será un entero.

  Los valores que retornará podrán ser:

  - -1: La conexión no se ha cerrado correctamente.
  - 1: La conexión se ha podido cerrar correctamente.

✓ Si lo que se desea es cerrar la conexión bruscamente, sin tener en cuenta los datos que estén pendientes de ser enviados ni tener que espera la confirmación de cierre de conexión por parte del host remoto, se llamará a *mti_abort*.

- ***int mti_abort (int numero_de_conexion):*** Abortará la conexión cerrándola bruscamente, sin esperar a que se envíen todos los datos que estaban en la cola. Al abortar se borran todas estructuras de la cola con datos a enviar a esta conexión. Al host remoto se le envía un paquete de reset para informar que se ha abortado la conexión.

  El parámetro que necesita es el siguiente:

  - int numero_de_conexión: Número entero que identifica la conexión que se quiere abortar.

  Al igual que en el caso en el que se cierra la conexión correctamente, los valores que retornará podrán ser:

  - -1: La conexión no se ha abortado correctamente.

- 1: La conexión se ha podido abortar correctamente.

8.2 ANEXO 2: TARJETAS HARDWARE

En este proyecto se pretende realizar una pila TCP/IP para poder comunicar varios dispositivos ya sean PCs, microcontroladores o PC microcontrolador.

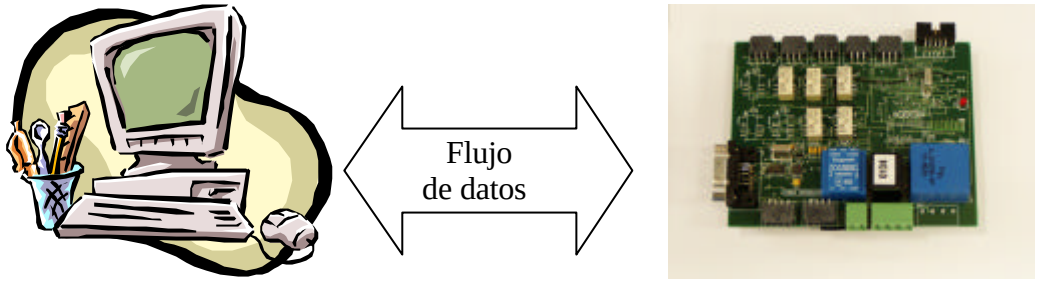

Figura 7.1

En el material utilizado se encuentra la tarjeta principal que contendrá al microcontrolador en el que se cargará el programa. Esta tarjeta se puede conectar con otra de las mismas características a través de fibra óptica. Para ello no sería necesaria ninguna placa más. También se podría conectar a un PC. Para ello será necesario incluir una tarjeta más que convierta la señal de salida del microcontrolador a una señal que pueda detectar el PC.

Nota: Estas tarjetas ya están diseñadas, en este punto se comentan las características más importantes que pueden hacer más fácil el entendimiento del proyecto.

- Placa principal:
  - En ella está instalado el microcontrolador donde se cargará el programa. El esquema de éste es el siguiente.

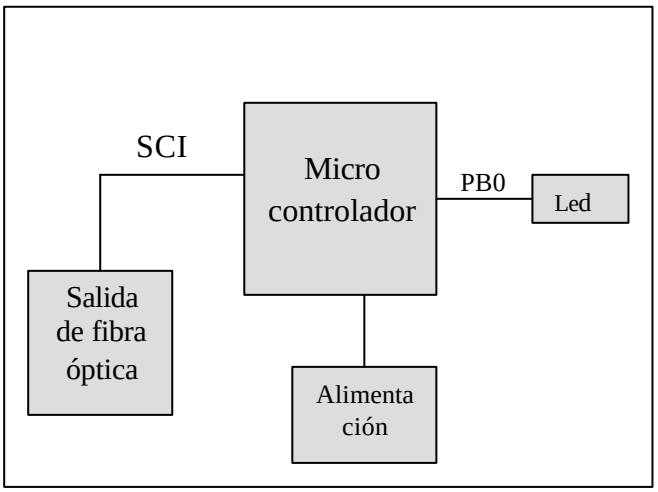

Figura 7.2

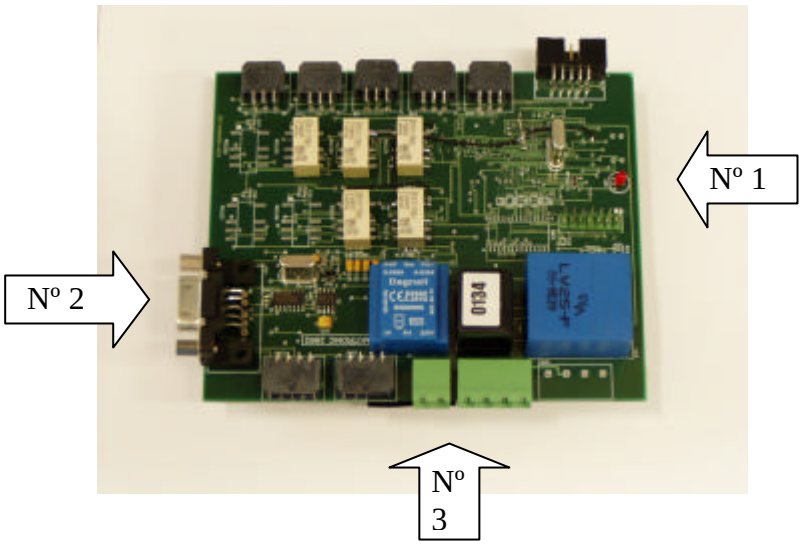

Figura 7.3

- La flecha número 1 apunta a la pata PB0 donde estará conectado el diodo LED que ayudará a la depuración del programa.
- La flecha número 2 apunta a la entrada/salida de fibra óptica. Ésta está conectada al puerto serie asíncrono SCI1 del microcontrolador y por ello los datos tendrán que ser adaptados para que la salida del SCI se transforme a una señal de fibra óptica y viceversa.
- La alimentación del microcontrolador será de 3.3 voltios.
- Otro dispositivo instalado en esta tarjeta es el Translator. Su función es la de transformar la señal de 5 voltios que proviene de la fibra a 3.3 voltios que es la que entenderá el microcontrolador.
- La flecha número 3 apunta a la entrada de la alimentación. Será necesario un regulador para adaptar dicha alimentación de entrada a la alimentación que necesita el microcontrolador.

- Placa secundaria: Convierte una señal proveniente de fibra óptica a una señal que cumpla la norma RS232 para que pueda ser leída por el PC.

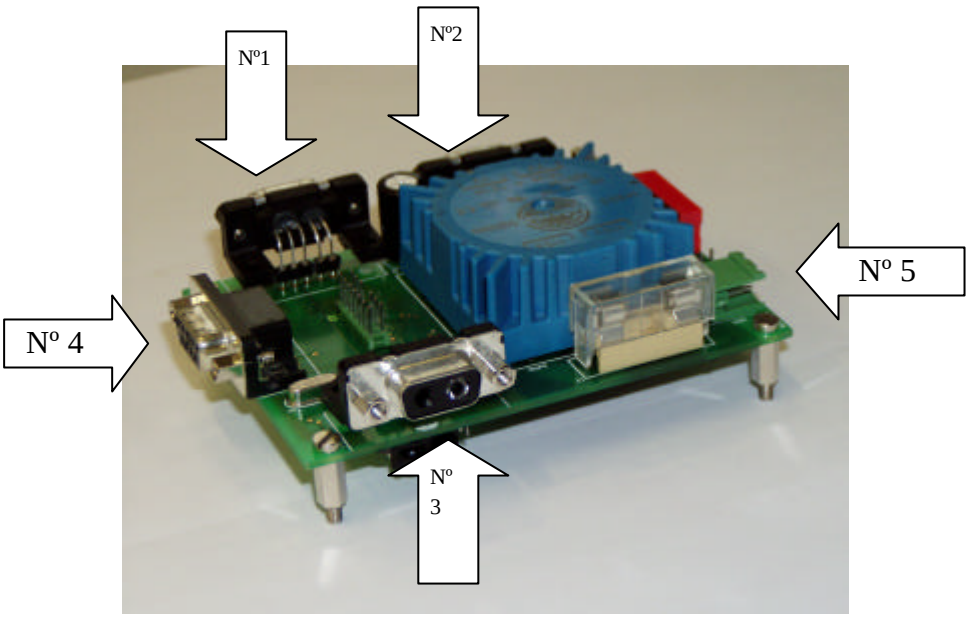

Figura 7.4

- Las flechas número 1,2 y 3 indican donde están las salidas de fibra óptica.
- La flecha número 4 indica la entrada procedente del puerto serie RS232 del PC.
- Por último la flecha número 5 apunta a la entrada de la alimentación.
- La señal procedente del puerto serie del PC se transformará a señal de fibra óptica y se transmitirá a todas las salidas de fibra. La señal que entre por cualquiera de las entradas de fibra óptica se convertirá a señal RS23.

8.3 ANEXO 3: PUERTO SERIE DEL PC

El puerto serie del PC es una buena herramienta para comunicarse con otros dispositivos o placas que necesiten intercambiar información. Por este motivo, a continuación se dan unas nociones sobre el funcionamiento de este puerto así como los registros más importantes y el código utilizado.

En un primer punto se introducirá el funcionamiento del puerto y posteriormente se describirán todos los registros que son necesarios así como su programación. Por último se describirán las dos funciones necesarias para la recepción y transmisión de datos.

8.3.1 INTRODUCCIÓN

Los ordenadores compatibles o PC tienen una salida al exterior llamada "puerto serie". El canal serie del PC es uno de los recursos más comunes para la conexión de periféricos, como pueden ser dispositivos de puntero (mouse) o de comunicación.

8.3.2 PUERTO SERIE DEL PC

El puerto serie se encarga de transmitir y/o recibir bytes o caracteres para poder comunicarse con otros equipos, ya sean otros ordenadores, como máquinas de diversa índole, fotocopiadoras, máquinas de análisis químicos, etc. De esta forma ya no sólo se dispone de disquetes, teclado o CD-ROM para poder introducir y sacar datos. El puerto serie lleva asociada una norma conocida como "comunicación serie síncrona/asíncrona". Esto quiere decir que los datos se envían uno detrás de otro y, por tanto, pueden ir solamente en un hilo eléctrico, de ahí que con sólo dos hilos o cables puedan transmitirse los datos y que de esta forma sea relativamente sencillo la comunicación.

El chip SERIE que incorpora el PC es el llamado ACE 8250, es un Elemento/chip de comunicación asíncrona (ACE), que está compuesto de una UART (Universal Asynchronous Receiver/Transmitter) ó Receptor/transmisor universal asíncrono y un BRG (Baud Rate Generator) ó Generador de baudios. Este chip soporta velocidades de hasta 625000 baudios, pero el BRG divide esta frecuencia para adaptarse al estándar RS-232C. Existe otro chip de comunicaciones llamado 16550 que es más moderno y potente.

La siguiente tabla muestra el ejemplo de 4 puertos COM típicos. Lo más normal es que tanto el COM1 como el COM2 estén ahora integrados en la placa base, o puestos en una tarjeta controladores. Los otros dos se suelen configurar con el modem (interno):

COM	Dirección base	IRQ
COM1	3F8	4
COM2	2F8	3
COM3	3E8	4
COM4	2E8	3

Tabla 7.1

Hay que tener en cuenta que el COM1 y el COM3 comparten la misma IRQ. Lo mismo sucede con COM2 y COM4.

El modo de transmitir los bits por el puerto serie es el siguiente:

- En primer lugar se enviará un bit de comienzo o bit de start indicando que empieza a transmitir un dato.
- A continuación se transmiten los bits de datos del menos significativo al más significativo. Los bits de datos pueden ser de 5 a 8.

- El siguiente bit enviado corresponde con el bit de paridad. Éste indica el estado de los bits que se han enviado/recibido. Podrá no existir paridad si no quiere chequearse este bit, o será par ó impar dependiendo del número de bits 1 ó 0 que se vean implicados.
- Por último se envía un bit de parada llamado bit de stop.

Ya que los bytes o caracteres pueden aparecer en cualquier momento, el receptor solamente se sincroniza con el emisor cuando recibe el bit de comienzo, por tanto, la velocidad de los dos equipos en su puerto serie debe ser la misma.

8.3.3 NORMA RS-232C

Esta norma cubre los tres aspectos siguientes, entre el equipo terminal de datos (DTE), y el equipo de comunicación de datos (DCE):

- Las características eléctricas de las señales.
- Las conexiones mecánicas de los conectores.
- La descripción funcional de las señales usadas.

La especificación mecánica considera un conector de 25 pines, con todas sus dimensiones bien especificadas. La especificación eléctrica considera que, para decidir si un bit está en 1, se deberá tener un voltaje más negativo que -3 volts; y para el bit 0, el voltaje sea superior a +4 volts.

Es posible tener velocidades de hasta 20 Kbps y longitud máxima de 15 metros de cable. La especificación funcional indica los circuitos que están conectados a cada uno de los 25 pines, así como el significado de cada uno de ellos de ellos.

8.3.4 PROGRAMACIÓN DEL PUERTO SERIE

8.3.4.1 Registros

Cada uno de los puertos COM posee varios registros:

- Registro base+0: tiene 3 funciones:
  - Transmitter Holding Register (THR): Se encarga de transmitir un dato (palabra) por el puerto.
  - Reciver Buffer Register (RBR): Su función será la de recibir un dato (palabra) del puerto.
  - Baud Rate Divisor Low (BRDL): Velocidad del canal, parte Baja.

- Registro base+1: tiene 2 funciones:
  - Baud Rate Divisor High (BRDH): Velocidad del canal, parte Alta.
  - Interrupt Enable Register (IER): Su función será la de activar o desactivar las interrupciones para el puerto COM.

- Registro base+2:
  - Interrupt ID Register (IIR): Controla la prioridad de las interrupciones

- Registro base+3:
  - Line Control Register (LCR): Controla los parámetros de configuración del canal serie (velocidad...)

- Registro base+4:
  - Modem Control Register (MCR): Activa las señales del modem.

- Registro base+5:
  - Line Status Register (LSR): Muestra el estado del canal serie (errores, etc.)

- Registro base+6:
  - Modem Status Register (MSR): Muestra el estado del modem.

A continuación se estudiarán los registros que serán utilizados en la configuración el puerto:

8.3.4.1.1 LCR (Line Control Register)

Con este registro se definen (configuran) los parámetros de la comunicación serie. DLAB será el bit de más peso y Num(1) el de menos.

DLAB	Break	Stick	Tipo	Paridad	Stop	Num(2)	Num(1)

Figura 8.1

- Num (1 y 2): Indica el numero de bits de datos en cada palabra que se van a utilizar. Se configura con la siguiente tabla:

Num2	Num1	Nº Bits
0	0	5
0	1	6
1	0	7
1	1	8

Tabla 7.2

- Stop: Indica el numero de bits de stop que enviara (o esperara) el puerto. Lo normal será colocar un 0 en este bit para que sólo se obtenga un bit de Stop. Si se pusiera un 1 y el número de bits de la palabra (configurada en num) fuera 5, el canal usara 1,5 bits de stop. En caso de que fuera mayor de 5, se usarían 2 Stop bits.

- Paridad: Indica si hay paridad en la comunicación serie. Con el bit igual a 0 indica que no existe paridad. Si el bit es igual a 1 indicará que sí existe. La paridad es un método para detectar errores.

- Tipo: Este bit indica el tipo de Paridad, si existiera (Configurada en el bit anterior), que se va a usar. Si el valor es un 0 indicará que la paridad es de modo impar, mientras que con un 1 será de modo par.

- Stick: Indica el nivel que se usará en la paridad: si este valor es un 0, se contarán el número de '1' para la paridad. Y si, en cambio, fuera un '1' se contarían el número de '0' para la paridad.

- Break: Fuerza un corte de la comunicación. Si se deja a 0 no pasara nada, pero si se pone a 1 cortará la comunicación y forzará la salida a '0'.

- DLAB: Bit interno. Inicialmente Se configurará este bit a 1 y, tras configurar la velocidad (los 2 registros), se volverá a configurar como 0.

  - Ejemplo:

    Si se quiere una configuración de 8 bits, 1 stop bit, sin paridad para un COM2 lo que habrá que hacer será:
    Primero se enviará a la dirección base más 3 (este caso como la dirección base del COM2 es 2F8,por tanto, 2F8+3 = 2FB) un 10000011 (0x83, con el DLAB a 1).
    Tras esto, se configurará la velocidad y se volverá a configurar el puerto pero con DLAB a 0 (00000011, 0x03).

8.3.4.1.2 LSR (Line Status Register)

Registro de estado de la línea. Suele ser el primer registro que se consulta tras una interrupción o una transmisión, ya que indica el estado de la línea. Éste es uno de los registros consultados cuando se accede al servicio 3 de la BIOS:

- BITS FUNCION
  - 0: Datos disponibles [Data ReadY]
  - 1: Error de sobrecarga [Overrun Error]
  - 2: Error de paridad [Parity Error]
  - 3: Error de tramas [Framing Error]
  - 4: Interrupción [Break Interrupt]
  - 5: Espera transmisión [Transm. Holding¦Register Empty]
  - 6: Transmisión [Transm. Empty]
  - 7: No se usa

8.3.4.1.3 MCR (Modem Control Register)

Es el encargado de controlar la interfaz con el modem, aunque no tiene por qué existir modem instalado, ya que pueden conectarse dos equipos mediante un cable, sin necesidad de instalación del modem, y es posible usar este registro para comprobar ciertas cosas, como por ejemplo la detección de portadora, etc.

- BITSFUNCION
  - 0: Terminal disponible [DTR]
  - 1: Esperando a envío [RTS]
  - 2: OUT1
  - 3: OUT2
  - 4: LOOP

8.3.4.1.4 BRSR (Baud Rate Select Register).

Este registro es el que selecciona la velocidad de transmisión. Está compuesto por dos registros de 8 bits llamados DLL y DLM que constituyen la parte baja y alta de un valor de 16 bits y que es un divisor que se aplica a la frecuencia base para seleccionar la

velocidad. La frecuencia base más utilizada por los PCs es de 1.8432MHz. El cálculo del divisor es el siguiente:

$$\text{Divisor} = \frac{\text{frec. base}}{16 \times \text{velocidad}}$$

Por tanto, si se quisiera utilizar una velocidad de 2400pbs, habría que pasarle el valor 1843200/(16*2400) = 48 al registro BRSR, o sea, DLL = 48 y DLM = 0. Hay veces en las que se quiere saber la velocidad en base al divisor, por ejemplo para un programa que lea la velocidad del puerto, para eso se sustituirá el divisor por la velocidad y queda como:

$$\text{velocidad} = \frac{\text{frec. Base}}{16 \times \text{divisor}}$$

Como se puede observar, existen muchas más velocidades que las que comúnmente se conocen por la BIOS.

8.3.4.1.5 THR (Transmitter Holding Register)

Este registro es de retención para la transmisión. Cuando se almacena un carácter en este registro, el registro interno de desplazamiento de transmisión comienza a enviar dicho carácter, cuando este registro de desplazamiento queda vacío volverá a cargarse con el que hay en el THR y al quedar vacío este registro, el bit THRE del LSR se activa (para que se pueda introducir el siguiente). Cuando están vacíos THR y el registro de desplazamiento, el bit TEMT del LSR también se activa.

8.3.4.1.6 RBR (Receiver Buffer Register)

A este registro se accede cuando hay un carácter esperando para leerse, esto se indica en el bit DR del LSR. El 8250 se encarga de que no se pierda este dato, aunque si aparece otro dato en este registro sin que se recogiese el anterior se generaría un error de sobrecarga o "overrun" bit OE del LSR.

8.3.4.1.7 IER (Interrupt Enable Register)

En este registro se seleccionan las interrupciones que quieren activarse cuando se produzcan las determinadas acciones. Deshabilitar el sistema de interrupciones como es lógico, desactiva / inhibe el IIR.

Bit	
0	Int. de dato disponible
1	Int. de reg, de retención de transm. vacío
2	Int. de error de recepción
3	Int. ante cambio del MSR

Tabla 8.3

8.3.4.2 Configuración del puerto serie del PC

La configuración del chip se hace mediante el registro LCR en el que se da el formato de la trama de bits que se envían o reciben, este formato viene definido por la longitud de palabra (número de bits de datos que van a enviarse / recibirse), el bit de parada (stop bit) y su control de paridad. Los valores para realizar este proceso, así como sus bits asociados, han sido comentados en el apartado de anterior.

Para configurar la velocidad hay que acceder a los registros DLL y DLM que, juntos, forman una palabra de 16 bits llamada divisor de velocidad, la cual fue comentada cuando se describió el registro BRSR. Este es el aspecto que más cambia con respecto a la llamada de la BIOS para la configuración del puerto, ya que aquí se dispone de mayor control para la selección de velocidad a la que se quiere transmitir / recibir.

Para configurar estos datos es MUY IMPORTANTE que se desconecten las interrupciones, o sea, enviar un 0 al registro IER antes de modificar algún parámetro de los registros antes mencionados, sino el resultado normal es la caída del sistema.

Esto se hace con la siguiente función:

```
void config()     /*configuración del puerto serie*/
{
      outportb((COM1+3),0x83);      /*DLAB=1, 1 stop bit, 8 bits de
                              datos y no paridad*/

      /*para  configurar  115200  es  1843200/(16*velocidad).  Será
      0x0001*/

      outportb(COM1,1);       /*byte menos significativo 0x01*/
      outportb((COM1+1),0);   /*byte más significativo 0x00*/
      outportb((COM1+3),3);   /*DLAB=0*/
      outportb((COM1+1),0);   /*IER => prohíbe interrupciones*/
      outportb((COM1+4),0);   /*MCR => loop externo*/
}
```

8.3.4.3 Transmisión de datos

La transmisión en el 8250 consiste en el control de los registros THR y TSR (Registro de desplazamiento de la transmisión). Los datos se transmiten bit a bit. Sólo habrá que esperar a que el indicador del LCR llamado THRE esté activo y entonces introducir el carácter en el THR, una vez hecho esto, hay que esperar a que el THRE vuelva a activarse para poder enviar otro dato al THR, así sucesivamente.

La siguiente función transmite un dato:

```
void enviar(int x)
{
     while((inportb(COM1+5) & 0X60) != 0x60){
          printf("estamos esperando para enviar un
          dato");
     }
     outportb(COM1,x);
}
```

En primer lugar se lee el registro LSC con la orden inportb. Si no está preparado para transmitir se quedará en un bucle hasta que pueda transmitir. Una vez que ya puede, envía el dato con la orden outportb.

8.3.4.4 Recibir datos

Para recibir un dato, simplemente hay que esperar a que el RBR esté lleno, o sea, contenga el dato que se recibe. Cuando el RBR contiene el dato, el bit DR del LSR se activa. En el caso de que llegue otro dato al RBR antes de que sea retirado el anterior, se produce un error de sobrecarga (Overrun Error) que se activa mediante el bit OE del LSR, también se activaría el PE (Parity Error) si se produjese un error de paridad ó el FE (Framing Error) si no se detecta el bit de stop.

A continuación se observa una función que recibe un dato:

```
void recibir()
{
     while((inportb(COM1+5) &1) == 0){
          printf("estamos esperando datos");
     }
     while((inportb(COM1+5) &1) != 0){
          buffer1[first1] = inportb(COM1);
          printf("\tdatos=%02x",buffer1[first1]);
          first1=(first1+1)%mti_TAM_BUFFER1;
     }
}
```

8.4 ANEXO 3: CÓDIGO C

Se adjunta todo el código C que implementa el programa que anteriormente se explicó detalladamente.

- ***mti.h***

```
#include <stdlib.h>
#include "mmc2107.h"
#include "proces.h"

/*************************************************
*    Function Prototypes                         *
*************************************************/
int main(void);
void isr_PIT1_PIF(void);
void isr_SCI1_RDRF(void);
void isr_EPF0(void);
void mcore_init_system(void);

/*************************************************************/
/*funciones que se pueden llamar desde el cliente o el servidor*/
/*************************************************************/
void mti_init(INT8U cadena[]);

/*********************************************************/
int mti_connect(INT8U rem_addr[], INT16U rm_puerto);

/*********************************************************/
int mti_listen(INT8U local_addr[], INT16U local_port);

/*********************************************************/
int mti_accept(int);

/*********************************************************/
int mti_read(int, INT8U *, int *);

/*********************************************************/
int mti_write(int, INT8U *, INT16U);

/*********************************************************/
int mti_close(int);

/*********************************************************/
int mti_abort(int);

/*********************************************************/
/* Los estados TCP usados en mti_conex_tcpestadobanderas*/
#define CLOSED      0
#define SYN_RCVD    1
#define SYN_SENT    2
#define ESTABLISHED 3
#define FIN_WAIT_1  4
#define FIN_WAIT_2  5
#define CLOSING     6
#define TIME_WAIT   7
#define LAST_ACK    8
#define LISTEN        9
#define CLOSE_WAIT  10
#define TS_MASK     15
/*********************************************************/
#define TCP_FIN 0x01
```

```
#define TCP_SYN 0x02
#define TCP_RST 0x04
#define TCP_PSH 0x08
#define TCP_ACK 0x10
#define TCP_URG 0x20
/*******************************************************/
#define mti_bien 1
#define mti_mal -1
/*******************************************************/
#define mti_no 0
#define mti_si 1
/*******************************************************/
typedef struct mti_estructura_conex {
      int num_conex;                    /*numero de la conexion */
      int espera_close;          /*cuando en close_wait no hay mas datos que
                                 enviar y esta esperando un close estara a 1 */
      int enviar_fin;            /*se activa si hay que enviar un fin */
      int espera_respuesta;
      int enviar_ack;
      INT8U tcpestadobanderas;          /* TCP estado y banderas */
      INT16U lc_puerto, rm_puerto;      /* el puerto y el remoto */
      INT8U lc_ip_dir[4];                     /*mi direccion IP */
      INT8U rm_ip_dir[4]; /* la direccion IP del par remoto */
      INT8U rcv_nxt[4];   /*el numero de secuencia que esperamos recibir */
      INT8U snd_nxt[4];   /*El numero de secuencia ultimo que nos enviaron */
      INT8U ack_nxt[4];   /* El numero de secuencia que deberia se asentido
                                 por el proximo ACK del par */
      INT8U temp;                /* el temporizador de retransmisiones */
      INT8U cont_retrans;        /* Cuenta el numero de retransmisiones para
                                 un segmento particular */
      int temporizador1;         /*temporizador   para   comprobar   si   hay   que
                                 retramsmitir de nuevo la trama */
      int temporizador2;         /*temporizador para esperar en TIME_WAIT */

      struct mti_estructura_conex *proximo;  /*puntero al proximi elemento */
} MTI_ESTRUCTURA_CONEX;

extern MTI_ESTRUCTURA_CONEX *mti_conex;       /*apunta a la conexion en uso */
extern MTI_ESTRUCTURA_CONEX *mti_conexion;    /*apunta  a  un  lista  enlacado  con
                                              las  estructuras  de  las  conexiones
                                              activas */

/*Cinco estructuras que almacenaran las conexiones activas añadido
al no poder utilizar el malloc*/
extern MTI_ESTRUCTURA_CONEX mti_conexion1;
extern MTI_ESTRUCTURA_CONEX mti_conexion2;
extern MTI_ESTRUCTURA_CONEX mti_conexion3;
extern MTI_ESTRUCTURA_CONEX mti_conexion4;
extern MTI_ESTRUCTURA_CONEX mti_conexion5;

/*******************************************************************/

/*Estructura que contiene las cabeceras TCP e IP*/
typedef struct mti_tcpip_cab {
      /* cabecera IP */
   INT8U vhl;
   INT8U tipo;
   INT8U tam[2];
   INT8U ipid[2];
   INT8U ipoffset[2];
   INT8U ttl;
   INT8U proto;
   INT16U ipchksum;
   INT8U fuente_ip_dir[4];
   INT8U dest_ip_dir[4];

   /* cabecera TCP */
   INT16U puerto_fuente;
```

```
   INT16U puerto_dest;
   INT8U seqno[4];
   INT8U ackno[4];
   INT8U tcpoffset;
   INT8U banderas;
   INT8U wnd[2];
   INT16U tcpchksum;
   INT16U urgp;
   INT8U mti_datos[125];
} MTI_TCPIP_CAB;

/*apunta a la cabecera que recibimos o que enviamos */
extern MTI_TCPIP_CAB *MTI_BUF;
/*Array para almacenar las tramas que no son utilizadas */
extern MTI_TCPIP_CAB mti_BUFFER_PAQUETES[3];
extern MTI_TCPIP_CAB mti_trama_anterior;
extern MTI_TCPIP_CAB mti_trama_anterior;

/*********************************************************/
struct send {             /*estructura que almacena los datos a enviar */
       int num_conex;             /*numero de la conexion a la que
                                  queremos enviar los datos */
       int enviado;
       int tam;                   /*tamaño de los datos */
       INT8U info[125];           /*informacion que queremos enviar */
       struct send *proximo;      /*puntero a la siguiente
                                  estructura con datos a enviar */
};

typedef struct send SEND;
extern SEND *mti_datos_a_enviar; /*lista enlazada de datos que tenemos que enviar
                                  */

/*Cinco estructuras que almacenaran las conexiones activas añadido
 al no poder utilizar el malloc*/
extern SEND mti_datos_a_enviar1;
extern SEND mti_datos_a_enviar2;
extern SEND mti_datos_a_enviar3;
extern SEND mti_datos_a_enviar4;
extern SEND mti_datos_a_enviar5;

/*****************************************************************/
MTI_ESTRUCTURA_CONEX *mti_buscar_id_conex(int);

/*****************************************************************/
MTI_ESTRUCTURA_CONEX *mti_buscar_tcpestadobanderas(MTI_ESTRUCTURA_CONEX *,
                                          int);
/*****************************************************************/
int mti_buscar_conex_libre(void);

/*****************************************************************/
MTI_ESTRUCTURA_CONEX *mti_buscar_puerto(MTI_ESTRUCTURA_CONEX *, INT16U);

/*****************************************************************/
void mti_borrar_estructura(MTI_ESTRUCTURA_CONEX *);

/*****************************************************************/
MTI_ESTRUCTURA_CONEX *mti_malloc_conex(void);

/*****************************************************************/
void mti_free_conex(MTI_ESTRUCTURA_CONEX *);

/*****************************************************************/
SEND *mti_buscar_datoenviar(int x);

/*****************************************************************/
void mti_borrar_datoenviado(int x);
```

```
/*********************************************************************/
SEND *mti_malloc_enviar(void);

/*********************************************************************/
void mti_free_enviar(SEND *);

int mti_encontrada_conexion(MTI_ESTRUCTURA_CONEX *, INT8U rem_addr[], INT16U);
void recibetrama(unsigned char *, unsigned long);
int escribeptoserie(unsigned char *trama, unsigned long size);
```

- ***proces.h***

```
#include "mmc2107.h"
#include "cardio.h"

/*************************************************************
definimos las constantes que vamos a usar
*************************************************************/

#define mti_TAM_BUFFER1 500
/*numero maximo de conexiones que se pueden establecer.*/
#define mti_num_max_conex 5
#define mti_libre 0
#define mti_ocupado 1

extern INT8U MTI_IP_DIR0;
extern INT8U MTI_IP_DIR1;
extern INT8U MTI_IP_DIR2;
extern INT8U MTI_IP_DIR3;

#define TTL         255

#define IP_PROTO_ICMP   1
#define IP_PROTO_TCP    6

#define MTI_RTO 3

/*****************************************************************
variables externas
*****************************************************************/
extern INT16U mti_tam_dato;      /*tamaño del dato */
extern int mti_num_conexion[mti_num_max_conex];     /*indica si la conexion esta
                                                     libre u ocupada */
extern INT8U buffer1[mti_TAM_BUFFER1];  /*buffer de entrada */
extern INT16U last1;             /*apunta al primer elemento que debe salir */
extern INT16U first1;            /*indica el ultimo elemento que entro */

extern INT16U ipid;
extern INT8U This_LED;

/*****************************************************/
/*funciones*/
/*****************************/
void mti_enviar_trama(void);

/*****************************/
void mti_enviar_cabecera(void);

/*****************************/
void mti_convierte_formatoSCI(INT8U *);

/*****************************/
int mti_trama_completa_rx(int *);

/*****************************/
int mti_reset(void);
```

```
/*****************************/
void mti_actualizar_rcv_nxt(INT16U);

/*****************************/
void mti_actualizar_ack_nxt(INT16U);

/*****************************/
int mti_proceso_estados(void);

/*****************************/
void mti_fin_procesoestados(void);

/*****************************/
void mti_enviar_finack(void);

/*****************************/
void mti_enviar_ack(void);

/*****************************/
void mti_enviar_fin(void);

INT16U mti_checksum(INT8U *, INT16U);
INT16U mti_ipchksum(void);
INT16U mti_tcpchksum(void);

/*****************************/
void mti_guardar_trama(int);
void mti_trama_anter(void);
```

- ***mti_abort.c***

```
#include "mti.h"
#include <stdio.h>

/*****************MTI_ABORT************************************
Aborta un aconexion. Si existen datos pendientes de ser enviados
se borraran.
*****************************************************************/

int
mti_abort(int numero_de_conexion)
{

/*buscar la conexion que indica numero_de_conexion
meterlo en conexion*/

   SEND *aux;
   int i;
   MTI_ESTRUCTURA_CONEX *punt_conex;

   punt_conex = mti_buscar_id_conex(numero_de_conexion);
   if(punt_conex == NULL) {
      /*hay error porque no hemos encontrado ningun socket con este numero */
      return -1;
   }


   if(punt_conex->tcpestadobanderas == CLOSED) {
      /*error no se puede cerrar un proceso que esta en closed */
      return -1;
   }

   if(punt_conex->tcpestadobanderas == LISTEN) {
   /*no envia nada al otro extremo porque aun no se ha establecido la conexion */
      mti_borrar_estructura(punt_conex);
```

```
      return 1;
   }

   if(punt_conex->tcpestadobanderas == SYN_SENT) {
      /*tenemos que borrar todos los datos que esten a la espera de ser
         enviados al otro extremo */
      aux = mti_buscar_datoenviar(mti_conex->num_conex);
      while(aux != NULL) {
        mti_borrar_datoenviado(mti_conex->num_conex);
        aux = mti_buscar_datoenviar(mti_conex->num_conex);
      };
      /*ya no hay datos que esten a la espera de enviarse en esta conexion,
         por tanto no tenemos que borrar nada */

      punt_conex->tcpestadobanderas = CLOSED;
      mti_borrar_estructura(punt_conex);
      return 1;
   }

   if((punt_conex->tcpestadobanderas == SYN_RCVD) ||
      (punt_conex->tcpestadobanderas == ESTABLISHED) ||
      (punt_conex->tcpestadobanderas == FIN_WAIT_1) ||
      (punt_conex->tcpestadobanderas == FIN_WAIT_2) ||
      (punt_conex->tcpestadobanderas == CLOSE_WAIT)) {
      /*tenemos que borrar todos los datos que esten a la espera */
      aux = mti_buscar_datoenviar(mti_conex->num_conex);
      while(aux != NULL) {
        mti_borrar_datoenviado(mti_conex->num_conex);
        aux = mti_buscar_datoenviar(mti_conex->num_conex);
      };
      /*mandamos un paquete de reset */
      i = mti_reset();          /*enviamos un paquete reset */
      if(i <= 0) {
        return -1;
      } else {
        mti_borrar_estructura(punt_conex);
        return 1;
      }
   }

   if((punt_conex->tcpestadobanderas == CLOSING) ||
      (punt_conex->tcpestadobanderas == LAST_ACK) ||
      (punt_conex->tcpestadobanderas == TIME_WAIT)) {
      mti_borrar_estructura(punt_conex);
      aux = mti_buscar_datoenviar(mti_conex->num_conex);
      while(aux != NULL) {
        mti_borrar_datoenviado(mti_conex->num_conex);
        aux = mti_buscar_datoenviar(mti_conex->num_conex);
      };
      /*ya no hay datos que esten a la espera de enviarse en esta conexion,
         por tanto no tenemos que borrar nada */
      punt_conex->tcpestadobanderas = CLOSED;
      mti_borrar_estructura(punt_conex);
      return 1;
   }
   /*si no es ningun caso de los anteriores devolvera -1 porque es un error */
   return -1;
}

/**************MTI_RESET*************************************************
enviamos un paquete de reset
************************************************************************/
int
mti_reset()
{
   int c;
   int tmpport;
```

```
/*No enviamos un reset en respuesta a un reset*/

   if(MTI_BUF->banderas & TCP_RST) {
      /*no enviamos un reset como respuesta de un reset */
      return -1;
   }
   MTI_BUF->banderas = TCP_RST | TCP_ACK;
   mti_tam_dato = 0;      /*ponemos el tamaño a 0 porque no enviamos datos */
   MTI_BUF->tcpoffset = 5 << 4;  /*MTI_BUF->tcpoffset corresponde con la longitud
                                 de la cabecera de TCP y los bit reservados(por
                                 eso se desplaza) */

   /*tira los campos secuencia y asentimiento en la cabecera TCP */

   c = MTI_BUF->seqno[3];  /*INT8U alto del numero de secuencia */
   MTI_BUF->seqno[3] = MTI_BUF->ackno[3];
   MTI_BUF->ackno[3] = c;

   c = MTI_BUF->seqno[2];
   MTI_BUF->seqno[2] = MTI_BUF->ackno[2];
   MTI_BUF->ackno[2] = c;

   c = MTI_BUF->seqno[1];
   MTI_BUF->seqno[1] = MTI_BUF->ackno[1];
   MTI_BUF->ackno[1] = c;

   c = MTI_BUF->seqno[0];
   MTI_BUF->seqno[0] = MTI_BUF->ackno[0];
   MTI_BUF->ackno[0] = c;

/*Nosotros tambien tenemos que incrementar el numero de secuencia
que hemos asentido. Si desborda el INT8U menos signeficativo,
necesitamos propagar el acarreo al otros INT8Us.*/

   if(++MTI_BUF->ackno[3] == 0) {
      if(++MTI_BUF->ackno[2] == 0) {
        if(++MTI_BUF->ackno[1] == 0) {
           ++MTI_BUF->ackno[0];
        }
      }
   }

/*Intercambio del numero de puerto*/

   tmpport = MTI_BUF->puerto_fuente;
   MTI_BUF->puerto_fuente = MTI_BUF->puerto_dest;
   MTI_BUF->puerto_dest = tmpport;

/*Intercambio del las direcciones IP*/

   tmpport = MTI_BUF->dest_ip_dir[0];
   MTI_BUF->dest_ip_dir[0] = MTI_BUF->fuente_ip_dir[0];
   MTI_BUF->fuente_ip_dir[0] = tmpport;

   tmpport = MTI_BUF->dest_ip_dir[1];
   MTI_BUF->dest_ip_dir[1] = MTI_BUF->fuente_ip_dir[1];
   MTI_BUF->fuente_ip_dir[1] = tmpport;


/*Y enviamos el paquete de reset (RST)*/

   mti_enviar_cabecera();
   return 1;
}
```

- ***mti_ack_fin.c***

```
#include "mti.h"
#include <stdio.h>

/*************MTI_ENVIAR_ACK*****************
Envia un paquete sin datos y con un ACK
********************************************/

void
mti_enviar_ack()
{
   mti_conex->espera_respuesta = mti_no;
   MTI_BUF->banderas = TCP_ACK;
   mti_tam_dato = 0;             /*esta a cero porque no enviamos datos */
   mti_enviar_trama();
   return;
}

/************MTI_ENVIAR_FIN****************
Envia un paquete de FIN sin datos
********************************************/

void
mti_enviar_fin()
{
   MTI_BUF->banderas = TCP_FIN;
   mti_tam_dato = 0;             /*esta a cero porque no enviamos datos */
   mti_enviar_trama();
   return;
}

/**********MTI_ENVIAR_FINACK*****************
envia un paquete de FINACK
********************************************/

void
mti_enviar_finack()
{
   MTI_BUF->banderas = TCP_FIN | TCP_ACK;
   mti_tam_dato = 0;             /*esta a cero porque no enviamos datos */
   mti_enviar_trama();
   return;
}
```

- ***mti_actualizar_nxt.c***

```
#include "mti.h"
#include <stdio.h>

/**************MTI_ACTUALIZAR_RCV_NXT*******************
Actualiza el numero mti_conex->rcv_nxt con el numero
que se le pasa como parametro
*********************************************************/
void
mti_actualizar_rcv_nxt(INT16U n)
{
   mti_conex->rcv_nxt[3] += (n & 0xff);
   mti_conex->rcv_nxt[2] += (n >> 8);

   if(mti_conex->rcv_nxt[2] < (n >> 8)) {
      ++mti_conex->rcv_nxt[1];
      if(mti_conex->rcv_nxt[1] == 0) {
        ++mti_conex->rcv_nxt[0];
```

```
      }
   }


   if(mti_conex->rcv_nxt[3] < (n & 0xff)) {
      ++mti_conex->rcv_nxt[2];
      if(mti_conex->rcv_nxt[2] == 0) {
        ++mti_conex->rcv_nxt[1];
        if(mti_conex->rcv_nxt[1] == 0) {
           ++mti_conex->rcv_nxt[0];
        }
      }
   }
}

/***************MTI_ACTUALIZAR_ACK_NXT*******************
Actualiza el numero mti_conex->ack_nxt con el numero
que se le pasa como parametro
************************************************/
void
mti_actualizar_ack_nxt(INT16U n)
{
   mti_conex->ack_nxt[3] += (n & 0xff);
   mti_conex->ack_nxt[2] += (n >> 8);

   if(mti_conex->ack_nxt[2] < (n >> 8)) {
      ++mti_conex->ack_nxt[1];
      if(mti_conex->ack_nxt[1] == 0) {
        ++mti_conex->ack_nxt[0];
      }
   }


   if(mti_conex->ack_nxt[3] < (n & 0xff)) {
      ++mti_conex->ack_nxt[2];
      if(mti_conex->ack_nxt[2] == 0) {
        ++mti_conex->ack_nxt[1];
        if(mti_conex->ack_nxt[1] == 0) {
           ++mti_conex->ack_nxt[0];
        }
      }
   }
}
```

- ***mti_close.c***

```
#include "mti.h"
#include <stdio.h>

/***************MTI_CLOSE****************************
Se llama cuando se quiere cerrar una conexion. Si ya se esta
cerrando devuelve -1
*****************************************************/

int
mti_close(int numero_de_conexion)
{

/*buscar la conexion que indica numero_de_conexion
meterlo en conexion*/

   MTI_ESTRUCTURA_CONEX *punt_conex;

   punt_conex = mti_buscar_id_conex(numero_de_conexion);
   if(punt_conex == NULL) {
      /*hay error porque no hemos encontrado ningun socket con este numero */
```

```
      return -1;
   }

   if(punt_conex->tcpestadobanderas == CLOSED) {
      /*error no se puede cerrar un proceso que esta en closed */
      return -1;
   }

   if(punt_conex->tcpestadobanderas == SYN_SENT) {
      mti_borrar_estructura(punt_conex);
      /*cerramos la conexion estando en SYN_SENT */
      return 1;
   }
   /*Para los tres casos siguientes si no hubiera datos pendientes de enviar
      se enviaria el FIN, si si hay se envian y despues de envia el FIN */

   if(punt_conex->tcpestadobanderas == SYN_RCVD) {
      punt_conex->enviar_fin = mti_si;
      return 1;
   }

   if(punt_conex->tcpestadobanderas == ESTABLISHED) {
      punt_conex->enviar_fin = mti_si;
      return 1;
   }

   if(punt_conex->tcpestadobanderas == CLOSE_WAIT) {
      punt_conex->enviar_fin = mti_si;
      return 1;
   }

   /*en los casos siguientes de devuelve -1 porque la conexion ya se
      esta cerrando */
   if((punt_conex->tcpestadobanderas == FIN_WAIT_1)
      || (punt_conex->tcpestadobanderas == FIN_WAIT_2)) {
      return -1;
   }

   if((punt_conex->tcpestadobanderas == CLOSING)
      || (punt_conex->tcpestadobanderas == LAST_ACK)) {
      return -1;
   }

   if(punt_conex->tcpestadobanderas == TIME_WAIT) {
      return -1;
   } else {
      /*no es ningun caso */
      return -1;
   }
}
```

- ***mti_chksum.c***

```
#include "mti.h"
#include <stdio.h>

/****************MTI_CHECKSUM*********************************
Calcula el checksum general de los datos que se le pasan
*************************************************************/
INT16U
mti_checksum(INT8U * datos, INT16U tam)
{
   INT16U j;
   INT16U datos_rx[255];
   u4 sum = 0;
   INT16U i;
```

```
   INT8U b[2];

   for(j = 0; j < tam; j = j + 1) {
      b[0] = *(datos++);
      b[1] = *(datos++);
      datos_rx[j] = ((b[0] << 8) + b[1]);
   }


   for(i = 0; i < tam; i = i + 1) {
      sum += (unsigned long) (datos_rx[i]);
   }

   while(sum >> 16)
      sum = (sum & 0xffff) + (sum >> 16);
   return (INT16U) ~ sum;

}

/****************MTI_IPCHKSUM*********************************
Calcula el checksum de la cabecera IP
*****************************************************/
INT16U
mti_ipchksum()
{
   INT16U k;

   k = mti_checksum((INT8U *) MTI_BUF, 10);
   return k;
}



/****************MTI_TCPCHKSUM*********************************
Calcula el checksum TCP
*****************************************************/
INT16U
mti_tcpchksum()
{
   INT8U a[255];
   int i;
   INT16U t, k;

   a[0] = MTI_BUF->fuente_ip_dir[0];
   a[1] = MTI_BUF->fuente_ip_dir[1];
   a[2] = MTI_BUF->fuente_ip_dir[2];
   a[3] = MTI_BUF->fuente_ip_dir[3];

   a[4] = MTI_BUF->dest_ip_dir[0];
   a[5] = MTI_BUF->dest_ip_dir[1];
   a[6] = MTI_BUF->dest_ip_dir[2];
   a[7] = MTI_BUF->dest_ip_dir[3];

   a[8] = 0;
   a[9] = MTI_BUF->proto;
   a[10] = 0;
   a[11] = (20 + mti_tam_dato);

   a[12] = (((MTI_BUF->puerto_fuente) >> 8) & 0xff);
   a[13] = ((MTI_BUF->puerto_fuente) & 0xff);
   a[14] = (((MTI_BUF->puerto_dest) >> 8) & 0xff);
   a[15] = ((MTI_BUF->puerto_dest) & 0xff);
   a[16] = MTI_BUF->seqno[0];
   a[17] = MTI_BUF->seqno[1];
   a[18] = MTI_BUF->seqno[2];
   a[19] = MTI_BUF->seqno[3];
   a[20] = MTI_BUF->ackno[0];
   a[21] = MTI_BUF->ackno[1];
```

```
   a[22] = MTI_BUF->ackno[2];
   a[23] = MTI_BUF->ackno[3];
   a[24] = MTI_BUF->tcpoffset;
   a[25] = MTI_BUF->banderas;
   a[26] = MTI_BUF->wnd[0];
   a[27] = MTI_BUF->wnd[1];
   a[28] = 0;
   a[29] = 0;
   a[30] = (((MTI_BUF->urgp) >> 8) & 0xff);
   a[31] = ((MTI_BUF->urgp) & 0xff);

   if(mti_tam_dato > 0) {
      for(i = 0; i < mti_tam_dato; i++) {
        a[32 + i] = MTI_BUF->mti_datos[i];
      }
   }
   if((mti_tam_dato % 2) != 0) {
      a[32 + mti_tam_dato] = 0;
      t = (12 + 20 + mti_tam_dato + 1) / 2;
   } else {
      t = (12 + 20 + mti_tam_dato) / 2;
   }
   k = mti_checksum(a, t);

   return k;
}
```

- ***mti_enviar_trama.c***

```
#include "mti.h"
#include <stdio.h>

/********MTI_ENVIAR_TRAMA*********************************************
Tenemos que rellenar los campos de las cabeceras TCP e IP antes
calculando el checksum y finalmente enviando el paquete
*************************************************************/

void
mti_enviar_trama()
{

   MTI_BUF->tcpoffset = 5 << 4;  /*La longitud de la cabecera siempre sera en
                                   LONGINT16U
                                5. Se desplaza 4 porque tcpoffset es un INT8U donde
                                los 4 primeros bits son la longitud y los 4
                                siguientes estan reservados. */

   MTI_BUF->ackno[0] = mti_conex->rcv_nxt[0];
   MTI_BUF->ackno[1] = mti_conex->rcv_nxt[1];
   MTI_BUF->ackno[2] = mti_conex->rcv_nxt[2];
   MTI_BUF->ackno[3] = mti_conex->rcv_nxt[3];

   MTI_BUF->seqno[0] = mti_conex->snd_nxt[0];
   MTI_BUF->seqno[1] = mti_conex->snd_nxt[1];
   MTI_BUF->seqno[2] = mti_conex->snd_nxt[2];
   MTI_BUF->seqno[3] = mti_conex->snd_nxt[3];

   MTI_BUF->puerto_fuente = mti_conex->lc_puerto;
   MTI_BUF->puerto_dest = mti_conex->rm_puerto;
   MTI_BUF->fuente_ip_dir[0] = mti_conex->lc_ip_dir[0];
   MTI_BUF->fuente_ip_dir[1] = mti_conex->lc_ip_dir[1];
   MTI_BUF->fuente_ip_dir[2] = mti_conex->lc_ip_dir[2];
   MTI_BUF->fuente_ip_dir[3] = mti_conex->lc_ip_dir[3];

   MTI_BUF->dest_ip_dir[0] = mti_conex->rm_ip_dir[0];
   MTI_BUF->dest_ip_dir[1] = mti_conex->rm_ip_dir[1];
```

```
   MTI_BUF->dest_ip_dir[2] = mti_conex->rm_ip_dir[2];
   MTI_BUF->dest_ip_dir[3] = mti_conex->rm_ip_dir[3];

   mti_enviar_cabecera();
   return;
}

/**********MTI_ENVIAR_CABECERA**********************************
rellena los parametros que han faltado anteriormente. Se hace asi
porque a veces al mandar un paquete no es necesario actualizar el
numero de secuencia y solo se llamaria a esta funcion
***************************************************************/

void
mti_enviar_cabecera()
{
   int i, z;

   INT8U mti_datos_SCI[255];

   MTI_BUF->vhl = 0x45;
   MTI_BUF->tipo = 0;
   MTI_BUF->ipid[0] = 0;
   MTI_BUF->ipid[1] = 0;
   MTI_BUF->ipoffset[0] = MTI_BUF->ipoffset[1] = 0;
   MTI_BUF->ttl = TTL;
   MTI_BUF->proto = IP_PROTO_TCP;
   MTI_BUF->wnd[0] = (mti_tam_dato >> 8);
   MTI_BUF->wnd[1] = (mti_tam_dato & 0xff);
   MTI_BUF->urgp = 0;
   mti_tam_dato = mti_tam_dato + 40;    /*le sumamos 40(longitud de la cabecera)a
                                          la
                                    longitud del dato para ponerlo en tamaño.
                                    despues se lo restaremos para tener solo
                                    la longitud del dato */
   MTI_BUF->tam[0] = (mti_tam_dato >> 8);
   MTI_BUF->tam[1] = (mti_tam_dato & 0xff);
   mti_tam_dato = mti_tam_dato - 40;

   /*    Calculamos los checksum de TCP e IP */
   MTI_BUF->ipchksum = 0;
   MTI_BUF->ipchksum = mti_ipchksum();
   MTI_BUF->tcpchksum = 0;
   MTI_BUF->tcpchksum = mti_tcpchksum();
   /*convierte a una cadena de octetos para que pueda ser enviada por el SCI */
   mti_convierte_formatoSCI(mti_datos_SCI);
   /*guarda la trama por si hay que reenviar */
   mti_trama_anter();

   enviar_SCI1(Baud_115200, mti_datos_SCI, 41);
   if(mti_tam_dato >= 1) {
      enviar_SCI1(Baud_115200, &MTI_BUF->mti_datos[0], mti_tam_dato);
   }
   MTI_BUF->puerto_dest = 0;
   return;
}

/*MTI_CONVIERTE_FORMATOSCI*****************************************
Convierte el buffer a una cadena de INT8U para pasarselo la SCI
*****************************************************************/

void
mti_convierte_formatoSCI(INT8U * datos_SCI)
{
   datos_SCI[0] = 0x11;
   datos_SCI[1] = MTI_BUF->vhl;
   datos_SCI[2] = MTI_BUF->tipo;
   datos_SCI[3] = MTI_BUF->tam[0];
```

```
   datos_SCI[4] = MTI_BUF->tam[1];
   datos_SCI[5] = MTI_BUF->ipid[0];
   datos_SCI[6] = MTI_BUF->ipid[1];
   datos_SCI[7] = MTI_BUF->ipoffset[0];
   datos_SCI[8] = MTI_BUF->ipoffset[1];
   datos_SCI[9] = MTI_BUF->ttl;
   datos_SCI[10] = MTI_BUF->proto;
   datos_SCI[11] = (((MTI_BUF->ipchksum) >> 8) & 0xff);
   datos_SCI[12] = ((MTI_BUF->ipchksum) & 0xff);
   datos_SCI[13] = MTI_BUF->fuente_ip_dir[0];
   datos_SCI[14] = MTI_BUF->fuente_ip_dir[1];
   datos_SCI[15] = MTI_BUF->fuente_ip_dir[2];
   datos_SCI[16] = MTI_BUF->fuente_ip_dir[3];
   datos_SCI[17] = MTI_BUF->dest_ip_dir[0];
   datos_SCI[18] = MTI_BUF->dest_ip_dir[1];
   datos_SCI[19] = MTI_BUF->dest_ip_dir[2];
   datos_SCI[20] = MTI_BUF->dest_ip_dir[3];
   datos_SCI[21] = (((MTI_BUF->puerto_fuente) >> 8) & 0xff);
   datos_SCI[22] = ((MTI_BUF->puerto_fuente) & 0xff);
   datos_SCI[23] = (((MTI_BUF->puerto_dest) >> 8) & 0xff);
   datos_SCI[24] = ((MTI_BUF->puerto_dest) & 0xff);
   datos_SCI[25] = MTI_BUF->seqno[0];
   datos_SCI[26] = MTI_BUF->seqno[1];
   datos_SCI[27] = MTI_BUF->seqno[2];
   datos_SCI[28] = MTI_BUF->seqno[3];
   datos_SCI[29] = MTI_BUF->ackno[0];
   datos_SCI[30] = MTI_BUF->ackno[1];
   datos_SCI[31] = MTI_BUF->ackno[2];
   datos_SCI[32] = MTI_BUF->ackno[3];
   datos_SCI[33] = MTI_BUF->tcpoffset;
   datos_SCI[34] = MTI_BUF->banderas;
   datos_SCI[35] = MTI_BUF->wnd[0];
   datos_SCI[36] = MTI_BUF->wnd[1];
   datos_SCI[37] = (((MTI_BUF->tcpchksum) >> 8) & 0xff);
   datos_SCI[38] = ((MTI_BUF->tcpchksum) & 0xff);
   datos_SCI[39] = (((MTI_BUF->urgp) >> 8) & 0xff);
   datos_SCI[40] = ((MTI_BUF->urgp) & 0xff);
}
```

- ***mti_guardar_trama.c***

```
#include "mti.h"
#include <stdio.h>

/************MTI_GUARDA_TRAMA*************************************
Cuando llega una trama, si no es para el puerto que esta nuestra
direccion IP pero no para el puerto que esta a la escucha,
tendremos que almacenarla para que otro puerto la pueda utilizar
*****************************************************************/

void
mti_guardar_trama(int i)
{
   INT16U tamano;
   int j;

   mti_BUFFER_PAQUETES[i].vhl = MTI_BUF->vhl;
   mti_BUFFER_PAQUETES[i].tipo = MTI_BUF->tipo;
   mti_BUFFER_PAQUETES[i].tam[0] = MTI_BUF->tam[0];
   mti_BUFFER_PAQUETES[i].tam[1] = MTI_BUF->tam[1];
   mti_BUFFER_PAQUETES[i].ipid[0] = MTI_BUF->ipid[0];
   mti_BUFFER_PAQUETES[i].ipoffset[0] = MTI_BUF->ipoffset[0];
   mti_BUFFER_PAQUETES[i].ipoffset[1] = MTI_BUF->ipoffset[1];
   mti_BUFFER_PAQUETES[i].proto = MTI_BUF->proto;
   mti_BUFFER_PAQUETES[i].ipchksum = MTI_BUF->ipchksum;
   mti_BUFFER_PAQUETES[i].fuente_ip_dir[0] = MTI_BUF->fuente_ip_dir[0];
```

```
   mti_BUFFER_PAQUETES[i].fuente_ip_dir[1] = MTI_BUF->fuente_ip_dir[1];
   mti_BUFFER_PAQUETES[i].fuente_ip_dir[2] = MTI_BUF->fuente_ip_dir[2];
   mti_BUFFER_PAQUETES[i].fuente_ip_dir[3] = MTI_BUF->fuente_ip_dir[3];
   mti_BUFFER_PAQUETES[i].dest_ip_dir[0] = MTI_BUF->dest_ip_dir[0];
   mti_BUFFER_PAQUETES[i].dest_ip_dir[1] = MTI_BUF->dest_ip_dir[1];
   mti_BUFFER_PAQUETES[i].dest_ip_dir[2] = MTI_BUF->dest_ip_dir[2];
   mti_BUFFER_PAQUETES[i].dest_ip_dir[3] = MTI_BUF->dest_ip_dir[3];
   mti_BUFFER_PAQUETES[i].puerto_fuente = MTI_BUF->puerto_fuente;
   mti_BUFFER_PAQUETES[i].puerto_dest = MTI_BUF->puerto_dest;
   mti_BUFFER_PAQUETES[i].seqno[0] = MTI_BUF->seqno[0];
   mti_BUFFER_PAQUETES[i].seqno[1] = MTI_BUF->seqno[1];
   mti_BUFFER_PAQUETES[i].seqno[2] = MTI_BUF->seqno[2];
   mti_BUFFER_PAQUETES[i].seqno[3] = MTI_BUF->seqno[3];
   mti_BUFFER_PAQUETES[i].ackno[0] = MTI_BUF->ackno[0];
   mti_BUFFER_PAQUETES[i].ackno[1] = MTI_BUF->ackno[1];
   mti_BUFFER_PAQUETES[i].ackno[2] = MTI_BUF->ackno[2];
   mti_BUFFER_PAQUETES[i].ackno[3] = MTI_BUF->ackno[3];
   mti_BUFFER_PAQUETES[i].tcpoffset = MTI_BUF->tcpoffset;
   mti_BUFFER_PAQUETES[i].banderas = MTI_BUF->banderas;
   mti_BUFFER_PAQUETES[i].wnd[0] = MTI_BUF->wnd[0];
   mti_BUFFER_PAQUETES[i].wnd[1] = MTI_BUF->wnd[1];
   mti_BUFFER_PAQUETES[i].tcpchksum = MTI_BUF->tcpchksum;
   mti_BUFFER_PAQUETES[i].urgp = MTI_BUF->urgp;
   tamano =
    (((mti_BUFFER_PAQUETES[i].tam[0] << 8) & 0xff00) +
     mti_BUFFER_PAQUETES[i].tam[1]);
   for(j = 0; j < tamano; j++) {
      mti_BUFFER_PAQUETES[i].mti_datos[j] = MTI_BUF->mti_datos[j];
   }
}

/********MTI_ENVIAR_TRAMA*********************************************
Tenemos que rellenar los campos de las cabeceras TCP e IP antes
calculando el checksum y finalmente enviando el paquete
**********************************************************************/

void
mti_trama_anter()
{
   int i;
   INT16U tam;

   mti_trama_anterior.vhl = MTI_BUF->vhl;
   mti_trama_anterior.tipo = MTI_BUF->tipo;
   mti_trama_anterior.tam[0] = MTI_BUF->tam[0];
   mti_trama_anterior.tam[1] = MTI_BUF->tam[1];
   mti_trama_anterior.ipid[0] = MTI_BUF->ipid[0];
   mti_trama_anterior.ipid[1] = MTI_BUF->ipid[1];
   mti_trama_anterior.ipoffset[0] = mti_trama_anterior.ipoffset[1] = 0;
   mti_trama_anterior.ttl = MTI_BUF->ttl;
   mti_trama_anterior.proto = MTI_BUF->proto;
   mti_trama_anterior.ipchksum = MTI_BUF->ipchksum;

   mti_trama_anterior.fuente_ip_dir[0] = MTI_BUF->fuente_ip_dir[0];
   mti_trama_anterior.fuente_ip_dir[1] = MTI_BUF->fuente_ip_dir[1];
   mti_trama_anterior.fuente_ip_dir[2] = MTI_BUF->fuente_ip_dir[2];
   mti_trama_anterior.fuente_ip_dir[3] = MTI_BUF->fuente_ip_dir[3];

   mti_trama_anterior.dest_ip_dir[0] = MTI_BUF->dest_ip_dir[0];
   mti_trama_anterior.dest_ip_dir[1] = MTI_BUF->dest_ip_dir[1];
   mti_trama_anterior.dest_ip_dir[2] = MTI_BUF->dest_ip_dir[2];
   mti_trama_anterior.dest_ip_dir[3] = MTI_BUF->dest_ip_dir[3];

   mti_trama_anterior.puerto_fuente = MTI_BUF->puerto_fuente;
   mti_trama_anterior.puerto_dest = MTI_BUF->puerto_dest;

   mti_trama_anterior.seqno[0] = MTI_BUF->seqno[0];
   mti_trama_anterior.seqno[1] = MTI_BUF->seqno[1];
```

```
   mti_trama_anterior.seqno[2] = MTI_BUF->seqno[2];
   mti_trama_anterior.seqno[3] = MTI_BUF->seqno[3];

   mti_trama_anterior.ackno[0] = MTI_BUF->ackno[0];
   mti_trama_anterior.ackno[1] = MTI_BUF->ackno[1];
   mti_trama_anterior.ackno[2] = MTI_BUF->ackno[2];
   mti_trama_anterior.ackno[3] = MTI_BUF->ackno[3];

   mti_trama_anterior.tcpoffset = MTI_BUF->tcpoffset;
   mti_trama_anterior.banderas = MTI_BUF->banderas;

   mti_trama_anterior.wnd[0] = MTI_BUF->wnd[0];
   mti_trama_anterior.wnd[1] = MTI_BUF->wnd[1];
   mti_trama_anterior.tcpchksum = MTI_BUF->tcpchksum;
   mti_trama_anterior.urgp = MTI_BUF->urgp;

   tam = ((mti_trama_anterior.tam[0] << 8)
        && 0xff00) + mti_trama_anterior.tam[1];
   for(i = 0; i < tam; i++)
      mti_trama_anterior.mti_datos[i] = MTI_BUF->mti_datos[i];
}
```

- ***mti_init.c***

```
#include "mti.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void mcore_init_system(void);

/**********VARIABLES GLOBALES*******************/

int mti_num_conexion[mti_num_max_conex];      /*indica si la conexion esta libre
                                               u ocupada */
INT16U mti_tam_dato;             /*la longitud 16 bits */
MTI_ESTRUCTURA_CONEX *mti_conex; /*apunta a la conexion en uso */

MTI_ESTRUCTURA_CONEX *mti_conexion;     /*Puntero a una estructura. Sera el
                                        principio de la lista dinamica donde se
                                        amacenan el estado de todas las
                                        conexiones */

/*Cinco estructuras que almacenaran las conexiones activas añadido al no
poder utilizar el malloc*/
MTI_ESTRUCTURA_CONEX mti_conexion1;
MTI_ESTRUCTURA_CONEX mti_conexion2;
MTI_ESTRUCTURA_CONEX mti_conexion3;
MTI_ESTRUCTURA_CONEX mti_conexion4;
MTI_ESTRUCTURA_CONEX mti_conexion5;

/*apunta al principio de las estructuras de los datos a enviar*/
SEND *mti_datos_a_enviar;

/*Cinco estructuras que almacenaran las conexiones activas añadido al no
poder utilizar el malloc*/
SEND mti_datos_a_enviar1;
SEND mti_datos_a_enviar2;
SEND mti_datos_a_enviar3;
SEND mti_datos_a_enviar4;
SEND mti_datos_a_enviar5;

static INT8U inic_numsec[4];     /*inicia el numero de secuencia */

/*Guarda la huella del último puerto usado  para una nueva conexión*/
static INT16U mti_ultimo_puerto;
```

```
/*guardara mi direccion IP*/
INT8U MTI_IP_DIR0;
INT8U MTI_IP_DIR1;
INT8U MTI_IP_DIR2;
INT8U MTI_IP_DIR3;

static int mti_num_listen;

MTI_TCPIP_CAB buf_entrada[1];
MTI_TCPIP_CAB *MTI_BUF;
MTI_TCPIP_CAB mti_trama_anterior;
MTI_TCPIP_CAB mti_BUFFER_PAQUETES[3];


static INT16U ipid;


/**********MTI_INIT*************************************************
inicia los punteros a NULL y guarda la direccion IP local en la variable global
*******************************************************************************/

void
mti_init(INT8U local_addr[])
{

   int i, c = 0;

   mti_ultimo_puerto = 0;

   mcore_init_system();
   /*inicializamos el array de las conexiones a libre. */
   for(c = 0; c < mti_num_max_conex; ++c) {
      mti_num_conexion[c] = mti_libre;
       }

   mti_num_listen = mti_num_max_conex + 1;    /*inicializamos a este valor para
                                               que no se confunda con una conexion
                                               */
   mti_conexion = NULL;          /*inicializamos el puntero a null porque no
                                 tenemos ninguna conexion */
   mti_datos_a_enviar = NULL;    /*inicializamos el puntero a null porque no
                                 tenemos ningun dato para enviar */
   inic_numsec[0] = 0;
   inic_numsec[1] = 0;
   inic_numsec[2] = 0;
   inic_numsec[3] = 1;

   MTI_TCPIP_CAB *MTI_BUF = &buf_entrada[0];

   /*añadido al no poder utilizar el malloc */
   mti_conexion1.num_conex = -1;
   mti_conexion2.num_conex = -1;
   mti_conexion3.num_conex = -1;
   mti_conexion4.num_conex = -1;
   mti_conexion5.num_conex = -1;

       mti_datos_a_enviar1.num_conex = -1;
   mti_datos_a_enviar2.num_conex = -1;
   mti_datos_a_enviar3.num_conex = -1;
   mti_datos_a_enviar4.num_conex = -1;
   mti_datos_a_enviar5.num_conex = -1;


   for(i = 0; i < 3; i++) {
      mti_BUFFER_PAQUETES[i].puerto_dest = 0;
   }
   MTI_IP_DIR0 = local_addr[0];
```

```
   MTI_IP_DIR1 = local_addr[1];
   MTI_IP_DIR2 = local_addr[2];
   MTI_IP_DIR3 = local_addr[3];

}


/************MTI_CONNECT*****************************************************
Devuelve un identificador de la conexión conectada a un puerto especificado
en rm_ip_dir(dirección ip remota).Si la conexión no se puede hacer, devuelve -1
*****************************************************************************/

int
mti_connect(INT8U rem_addr[], INT16U rm_puerto)
{
   int i = 0;
   int z = 0;

   /*Buscamos un puero local sin usar */
   MTI_ESTRUCTURA_CONEX *auxiliar;
   MTI_ESTRUCTURA_CONEX *paux;
   MTI_ESTRUCTURA_CONEX *conex1;

   auxiliar = mti_conexion;
       ++mti_ultimo_puerto;
   if(mti_ultimo_puerto >= 32767) {     /*si ya no hay mas puertos disponibles */
      mti_ultimo_puerto = 0;     /*volvemos a empezar por cero. */
   }


/*Caso 1: como al cerrar una conexion se borra la estructura que
mantiene el estado, el puerto esta en closed*/

   if(mti_conexion == NULL) {
      i = mti_buscar_conex_libre();
      if(i <= 0) {
        return -1;
      } else {                   /*reservamos memoria para esta estructura. */
        mti_conexion = &mti_conexion1;
        conex1 = mti_conexion;
        conex1->num_conex = i;
        conex1->proximo = NULL;
        z = mti_encontrada_conexion(conex1, rem_addr, rm_puerto);
        return z;
      }
   }


/*Caso 2:que el puerto almacenado en ultimo puerto ya este ocupado.
vamos aumentando ultimo puerto hasta que encontremos uno sin usar*/

   while(auxiliar != NULL) {
      conex1 = auxiliar;
      auxiliar = conex1->proximo;
      if(conex1->lc_puerto == mti_ultimo_puerto) {
        ++mti_ultimo_puerto;
        if(mti_ultimo_puerto >= 32767) {
           mti_ultimo_puerto = 0;
        }
      }
       }
   /*cuandotermina el while ya tenemos un numero de puerto asignado */

/*Caso 3:el puerto esta en time_wait y se puede utilizar.*/
   auxiliar = mti_conexion;
   conex1 = mti_buscar_tcpestadobanderas(auxiliar, TIME_WAIT);
   if(conex1 != NULL) {
      z = mti_encontrada_conexion(conex1, rem_addr, rm_puerto);
```

```
      return z;
   }

   auxiliar = mti_conexion;
   while(auxiliar != NULL) {
      paux = auxiliar;
      auxiliar = auxiliar->proximo;
   };
/*reservamos memoria para esta estructura.*/

       i = mti_buscar_conex_libre();
   if(i <= 0) {
      return -1;
   } else {
      paux->proximo = mti_malloc_conex();
      if(paux->proximo == NULL) {
        return -1;
      } else {
        conex1 = paux->proximo;
        conex1->num_conex = i;
        conex1->proximo = NULL;
        z = mti_encontrada_conexion(conex1, rem_addr, rm_puerto);
        return z;
      }
   }
}

/*****************************************************/
int
mti_encontrada_conexion(MTI_ESTRUCTURA_CONEX * conex1, INT8U rem_addr[],
                    INT16U rm_puerto)
{

/*el puerto encontrado no esta siendo utilizado, por lo tanto lo puedo utilizar
yo.*/

   conex1->snd_nxt[0] = conex1->ack_nxt[0] = inic_numsec[0];
   conex1->snd_nxt[1] = conex1->ack_nxt[1] = inic_numsec[1];
   conex1->snd_nxt[2] = conex1->ack_nxt[2] = inic_numsec[2];
   conex1->snd_nxt[3] = conex1->ack_nxt[3] = inic_numsec[3];

   if(++conex1->ack_nxt[3] == 0) {
      if(++conex1->ack_nxt[2] == 0) {
        if(++conex1->ack_nxt[1] == 0) {
           ++conex1->ack_nxt[0];
        }
      }
       }
/*INICIALIZAMOS RCV_NXT A CERO PORQUE HASTA QUE NO NOS ENVIEN EL PAQUETE
SYNACK NO SABEMOS CUAL ES ESTE VALOR.*/

   conex1->rcv_nxt[0] = 0;
   conex1->rcv_nxt[1] = 0;
   conex1->rcv_nxt[2] = 0;
   conex1->rcv_nxt[3] = 0;

   conex1->cont_retrans = 2;
   conex1->temp = MTI_RTO; /*El paquete de conexion solo se envia una vez */
   conex1->lc_puerto = mti_ultimo_puerto;
   conex1->rm_puerto = rm_puerto;

   conex1->lc_ip_dir[0] = MTI_IP_DIR0;
   conex1->lc_ip_dir[1] = MTI_IP_DIR1;
   conex1->lc_ip_dir[2] = MTI_IP_DIR2;
   conex1->lc_ip_dir[3] = MTI_IP_DIR3;

   conex1->rm_ip_dir[0] = rem_addr[0];
   conex1->rm_ip_dir[1] = rem_addr[1];
```

```
   conex1->rm_ip_dir[2] = rem_addr[2];
   conex1->rm_ip_dir[3] = rem_addr[3];

/*Como estoy trabajando directamente con los punteros a la estructura
ya quedan actualizados los campos en la lista.*/

/*una vez encontrada conexion, enviamos un SYN*/

   mti_tam_dato = 0;            /*lo ponemos a cero porque no estamos mandando
                                datos, solo una trama con un SYN */
   conex1->tcpestadobanderas = SYN_SENT;
   conex1->enviar_fin = mti_no;
   conex1->espera_close = mti_no;
   conex1->espera_respuesta = mti_si;
   MTI_BUF->banderas = TCP_SYN;
/*mti_conex es la conexion que esta en uso.*/
   mti_conex = conex1;
   mti_enviar_trama();

   return conex1->num_conex;
}

/*************MTI_LISTEN***************************************************
pone un puerto a la escucha. Crea una estructura con el estado en listen
*****************************************************************************/
int
mti_listen(INT8U local_addr[], INT16U local_port)
{
   MTI_ESTRUCTURA_CONEX *puntero;
   MTI_ESTRUCTURA_CONEX *auxiliar;
   MTI_ESTRUCTURA_CONEX *paux;

   /*primero se busca si el puerto ya esta siendo utilizado por otra conexion */
       puntero = mti_buscar_puerto(mti_conexion, local_port);
   while(puntero != NULL) {
      if((puntero->tcpestadobanderas != CLOSED) && (puntero->tcpestadobanderas !=
LISTEN)) {   /*este puerto ya esta siendo  usado */
        return -1;
      }
      if(puntero->tcpestadobanderas == LISTEN) {
        return 2;          /*el puerto ya esta a la escucha */
      }
      auxiliar = puntero->proximo;
      puntero = mti_buscar_puerto(auxiliar, local_port);
   }
   /*no hay ninguna conexion activa que utilice este puerto o si el puerto
   esta a la escucha para otra conexion podemos ponerlo a la escucha para esta */

   /*Es la primera estructura que se crea */
   if(mti_conexion == NULL) {
      mti_conexion = &mti_conexion1;
              puntero = mti_conexion;
   } else {               /*se busca una estructura libre */
      paux = mti_conexion;
      auxiliar = paux->proximo;
      while(auxiliar != NULL) {
        paux = auxiliar;
        auxiliar = auxiliar->proximo;
      };
      paux->proximo = mti_malloc_conex();
      if(paux->proximo == NULL) {
        return -1;
      } else {
        puntero = paux->proximo;
      }
   }
   mti_num_listen = mti_num_listen++;
```

```
   puntero->num_conex = mti_num_listen; /*este no es el numero de la conexion lo
                                        utilizamos para pasarselo a accept y alli
                                        se pondra el numero de conexion correcto
                                        */

   puntero->tcpestadobanderas = LISTEN;
   puntero->lc_ip_dir[0] = local_addr[0];
   puntero->lc_ip_dir[1] = local_addr[1];
   puntero->lc_ip_dir[2] = local_addr[2];
   puntero->lc_ip_dir[3] = local_addr[3];

   puntero->lc_puerto = local_port;
   puntero->enviar_fin = mti_no; /*se  activara  cuando  se  realize  una  llamada
                                 close */
   puntero->espera_close = mti_no;      /*estara a la espera de que el par envie
                                        un close */
   return puntero->num_conex;
}

/***************MTI_ACCEPT*********************************************
acepta una conexion cuando le llega un paquete de SYN
**********************************************************************/
int
mti_accept(int x)
{
   int i, z;
   int cont_trama_completa = 0;
   MTI_ESTRUCTURA_CONEX *puntero;
   MTI_ESTRUCTURA_CONEX *paux;

   /*si no ha puesto el puerto a la escucha no se puede aceptar una conexion */
   if(x == -1) {
      return -1;
   }

   mti_conex = mti_buscar_id_conex(x);
   if(mti_conex == NULL) {
      return -1;
   }

   if(mti_conex->tcpestadobanderas != LISTEN) {
             return -1;         /*existe  algun  error,  aunque  la  pusimos  a  la
                                escucha */
   }
   /*se busca si hay alguna conexion que se haya adelantado y haya pasado
      de LISTEN a otro estado antes que esta conexion y asi pues aqui no
      se puede usar este puerto. */
   puntero = mti_buscar_puerto(mti_conexion, mti_conex->lc_puerto);
   while(puntero != NULL) {
      if(puntero->num_conex != x) {
        if((puntero->tcpestadobanderas != LISTEN) &&
           (puntero->tcpestadobanderas != CLOSED)) {
           return -1;           /*el puerto ya esta siendo utilizado */
        }
      }
      paux = puntero->proximo;
      puntero = mti_buscar_puerto(paux, mti_conex->lc_puerto);
   }

   /*En primer lugar se busca si hay alguna trama almacenada que este destinada
       a este puerto local, si no es asi esperaremos para recibir otra trama */
   MTI_BUF = &mti_BUFFER_PAQUETES[0];
   if(MTI_BUF->puerto_dest != mti_conex->lc_puerto) {
      MTI_BUF = &mti_BUFFER_PAQUETES[1];
      if(MTI_BUF->puerto_dest != mti_conex->lc_puerto) {
        MTI_BUF = &mti_BUFFER_PAQUETES[2];
        if(MTI_BUF->puerto_dest != mti_conex->lc_puerto) {
           mti_conex->temporizador1 = 0;
```

```
        z = mti_trama_completa_rx(&cont_trama_completa);
        while(z == 0) {
           z = mti_trama_completa_rx(&cont_trama_completa);
           if(mti_conex->temporizador1 >= 200000) {
                 /*No puedo aceptar una conxion que si no me llega un SYN.
                 Espero un tiempo y si no ha llegado este paquete devolvere
                 error */
            mti_free_conex(mti_conex);
            return -1;
           }
        }
            /*se actualiza el contador a cero ya que si se ha pasado aqui es
           porque no hay que retransmitir mas esta trama */

        if(z == -1) {   /*hay error en la recepcion de la trama */
           return -1;
        }

        if(z == 2) {    /*el paquete no es para esta conexion */
           return 2;
        }
           /*Si ha llegado un paquete bien y no es para nuestro puerto tenemos
           que almacenarlo en el buffer para que otra conexion la pueda
           utilizar. Buscamos que haya espacio para almacenarlo y si no
           sobreescribimos */
        if(MTI_BUF->puerto_dest != mti_conex->lc_puerto) {
           if(mti_BUFFER_PAQUETES[0].puerto_fuente == 0) {
             /*meter MTI_BUF en mti_BUFFER_PAQUETES0 */
             mti_guardar_trama(0);
             return 3;
           } else if(mti_BUFFER_PAQUETES[1].puerto_fuente == 0) {
             /*meter MTI_BUF en mti_BUFFER_PAQUETES1 */
             mti_guardar_trama(1);
             return 3;
           } else if(mti_BUFFER_PAQUETES[2].puerto_fuente == 0) {
             /*meter MTI_BUF en mti_BUFFER_PAQUETES2 */
             mti_guardar_trama(2);
             return 3;
           } else {
             /*meter MTI_BUF en mti_BUFFER_PAQUETES0 */
             mti_guardar_trama(0);
             return 3;
           }
        }
     }
   }
}

if(MTI_BUF->banderas != TCP_SYN) {
   return -1;
}

/*lleqa un paquete con una conexion en LISTEN y que ha recibido un syn.
  enviamos un synack. Rellemanos los campos necesarios en la nueva conexion */

i = mti_buscar_conex_libre();
if(i <= 0) {
   mti_free_conex(mti_conex); /*habiamos creado una estructura cuando poniamos
                                el puerto a la escucha. como no hay conexion
                                libre habria que borrarla */
   return -1;
}
mti_conex->num_conex = i;
mti_conex->temp = MTI_RTO;    /*numero de veces que podremos retransmitir una
                               trama */
mti_conex->cont_retrans = 2;
mti_conex->espera_respuesta = mti_si;
```

```
   mti_conex->rm_puerto = MTI_BUF->puerto_fuente;

   mti_conex->rm_ip_dir[0] = MTI_BUF->fuente_ip_dir[0];
   mti_conex->rm_ip_dir[1] = MTI_BUF->fuente_ip_dir[1];
   mti_conex->rm_ip_dir[2] = MTI_BUF->fuente_ip_dir[2];
   mti_conex->rm_ip_dir[3] = MTI_BUF->fuente_ip_dir[3];

   mti_conex->tcpestadobanderas = SYN_RCVD;

   mti_conex->snd_nxt[0] = mti_conex->ack_nxt[0] = inic_numsec[0];
   mti_conex->snd_nxt[1] = mti_conex->ack_nxt[1] = inic_numsec[1];
   mti_conex->snd_nxt[2] = mti_conex->ack_nxt[2] = inic_numsec[2];
   mti_conex->snd_nxt[3] = mti_conex->ack_nxt[3] = inic_numsec[3];
   mti_actualizar_ack_nxt(1);

   /* se actualiza rcv_nxt incrementandolo en 1 */
   mti_conex->rcv_nxt[3] = MTI_BUF->seqno[3];
   mti_conex->rcv_nxt[2] = MTI_BUF->seqno[2];
   mti_conex->rcv_nxt[1] = MTI_BUF->seqno[1];
   mti_conex->rcv_nxt[0] = MTI_BUF->seqno[0];
   mti_actualizar_rcv_nxt(1);

   /*enviamos la respuesta a un SYN, es decir, enviamos un SYNACK */
   MTI_BUF->banderas = TCP_SYN | TCP_ACK;
   mti_enviar_trama();
   return 1;
}
```

- ***mti_malloc_free.c***

```
#include "mti.h"
#include <stdio.h>

/*********************MTI_MALLOC_CONEX*********************************
Busca una estructura que podamos utilizar para una nueva conexion
**********************************************************************/

MTI_ESTRUCTURA_CONEX *
mti_malloc_conex()
{
   MTI_ESTRUCTURA_CONEX *paux_malloc;

   if(mti_conexion1.num_conex == -1) {
      paux_malloc = &mti_conexion1;
   } else if(mti_conexion2.num_conex == -1) {
      paux_malloc = &mti_conexion2;
   } else if(mti_conexion3.num_conex == -1) {
      paux_malloc = &mti_conexion3;
   } else if(mti_conexion4.num_conex == -1) {
      paux_malloc = &mti_conexion4;
   } else if(mti_conexion5.num_conex == -1) {
      paux_malloc = &mti_conexion5;
   } else {
      paux_malloc = NULL;
   }
   return paux_malloc;
}

/***************MTI_FREE_CONEX*************************************
Limpia una estructura que ya no necesitamos para que otra conexion
la pueda utilizar
**********************************************************************/

void
mti_free_conex(MTI_ESTRUCTURA_CONEX * conexion)
{
```

```
   conexion->num_conex = -1;
}

/*********************MTI_MALLOC_ENVIAR*********************************
Busca una estructura que podamos utilizar para almacenar datos
*************************************************************************/

SEND *
mti_malloc_enviar()
{
   SEND *paux;

   if(mti_datos_a_enviar1.num_conex == -1) {
      paux = &mti_datos_a_enviar1;
   } else if(mti_datos_a_enviar2.num_conex == -1) {
      paux = &mti_datos_a_enviar2;
   } else if(mti_datos_a_enviar3.num_conex == -1) {
      paux = &mti_datos_a_enviar3;
   } else if(mti_datos_a_enviar4.num_conex == -1) {
      paux = &mti_datos_a_enviar4;
   } else if(mti_datos_a_enviar5.num_conex == -1) {
      paux = &mti_datos_a_enviar5;
   } else {
      paux = NULL;
   }
   return paux;
}

/***************MTI_FREE_ENVIAR*****************************************
Limpia una estructura que ya no necesitamos para que se puedan
almacenar otros datos
*************************************************************************/

void
mti_free_enviar(SEND * datos)
{
   datos->num_conex = -1;
}
```

- ***mti_puerto_serie.c***

```
#include "mti.h"
#include <stdio.h>

/***********************************************
*   Includes, Exports and External References
***********************************************/
#include "mmc2107.h"
#include <string.h>
#include <stdlib.h>
#include "typedefs.h"
#include "mmc2107_initvals.h"
#include "m2107_led.h"
#include "cardio.h"

/*************************************
macors para las interrupciones del MCORE
*************************************/
#define EnableInterrupts WritePSR(ReadPSR() | IE | EE)
#define DisableInterrupts WritePSR(ReadPSR() & ~IE & ~EE)

/***********************************
*    Variables globales del MCORE
***********************************/
INT8U This_LED;                 /* A flag to signify which LED to turn on */
INT16U first1, last1;           /* buffer1 index */
```

```
INT8U buffer1[mti_TAM_BUFFER1];  /* buffer to load the SCI1 data */
INT8U overflow1;           /* overflow in buffer2 */


/*************************************************************************
***************
Programa de interrupciones
*************************************************************************
**************/
#pragma interrupt on
/*La siguiente funcion se ejecuta cada vez que salta el temporizador*/
void
isr_PIT1_PIF(void)
{
   MTI_ESTRUCTURA_CONEX *paux_temp;

   mmc_clear_PIT1_int;           /* Clear the PIT1 interrupt flag */
   /*Esto lo defino yo. Es lo que quiero que haga cuando salte mi interrupcion */
   paux_temp = mti_conexion;
   while(paux_temp != NULL) {
      ++(paux_temp->temporizador1);
      ++(paux_temp->temporizador2);
      paux_temp = paux_temp->proximo;
       };
}

void
isr_EPF0(void)
{
   int i, flag;

   flag = 1;              /*initialize control flag */
   for(i = 0; i < 10 && flag == 1; i++) {
      if(reg_EPPDR.bit.EPPD0 == 1)
        flag = 0;          /* value incorrect */
   }

   if(flag == 1) {
      ;                          /* interrupt code */
   }

       mmc_clear_EPF0_int;
}


void
isr_SCI1_RDRF(void)
{

   /* Se machaca los datos antiguos en caso de overflow
      en caso de querer utilizarlo solo se tiene que
      quitar los simbolos de  comentario */

   INT8U temp;

   /* Se desechan los caracteres nuevos en caso de overflow */
   first1 = (first1 + 1) % mti_TAM_BUFFER1;
   temp = reg_SCI1SR1.reg; /*Limpia bit de interrupcion y lee el dato */
   if(first1 == last1 - 1) {
             overflow1 = 1;
      first1--;
      temp = reg_SCI2DRL;
   } else
      buffer1[first1] = reg_SCI1DRL;
}


#pragma interrupt off
```

```
/**************MCORE_INIT_SYSTEM*****************************************
Inicializa el micro
************************************************************************/

void
mcore_init_system()
{
   last1 = 1;
      first1 = 0;
   overflow1 = 0;
   This_LED = 0;
   /* Iniciamos todos los registros del micro */
   SystemInit();          /* Initialize all registers of MMC2107 */


   /* Rutinas      de configuración de los perifericos */
   PIT1(4, ON_DOZE, ENABLE_INT, ENABLE, 10000);
   SCI1(Baud_115200, ON_DOZE, ENABLE, ENABLE);

/* Habilitamos Interrupciones*/

   mmc_enNormalInterrupt1; /* Enable normal vectored interrupts, priority 1 */
   mmc_enNormalInterrupt2; /* Enable normal vectored interrupts, priority 2 */
    /*mmc_enNormalInterrupt3; *//* Enable normal vectored interrupts, priority 3
*/
   EnableInterrupts;             /* Enable interrupts at the core */

}
```

- ***mti_read.c***

```
#include "mti.h"
#include <stdio.h>

/************MTI_READ*****************************************
Lee lo que hay en el buffer de entrada y lo procesa
************************************************************/

int
mti_read(int socket, INT8U * buffer, int *tam_buffer)
{
   int i, k, z;
   SEND *punt_dato;
   INT8U mti_datos_SCI[255];
   int cont_trama_completa = 0;

   *tam_buffer = 0;

   mti_conex = mti_buscar_id_conex(socket);
   if(mti_conex == NULL) {
      return -1;
   }

   /*si estamos en close_wait y hemos enviado todos los datos pendientes
      esperamos a que cierren con close */
   if(mti_conex->tcpestadobanderas == CLOSE_WAIT) {
      if(mti_conex->espera_close == mti_si) {
        mti_conex->espera_close = mti_no;
        mti_actualizar_ack_nxt(1);
        mti_fin_procesoestados();
        return 1;
      }
   }
   punt_dato = mti_buscar_datoenviar(mti_conex->num_conex);
   if(punt_dato == NULL) {
```

```
      /*no hay datos para enviar */
   } else {
      punt_dato->enviado = mti_no;
      mti_fin_procesoestados();
      return 1;
   }

   if(mti_BUFFER_PAQUETES[0].puerto_dest == mti_conex->lc_puerto) {
      MTI_BUF = &mti_BUFFER_PAQUETES[0];
   } else if {
   mti_BUFFER_PAQUETES[1].puerto_dest == mti_conex->lc_puerto) {
      MTI_BUF = &mti_BUFFER_PAQUETES[1];
   } else if(mti_BUFFER_PAQUETES[2].puerto_dest == mti_conex->lc_puerto) {
      MTI_BUF = &mti_BUFFER_PAQUETES[2];
   } else {
      mti_conex->temporizador1 = 0;
      z = mti_trama_completa_rx(&cont_trama_completa);
      /*ESPERAMOS QUE NOS LLEGUE UNA TRAMA COMPLETA */
      while(z == 0) {
        z = mti_trama_completa_rx(&cont_trama_completa);
        if(mti_conex->temporizador1 >= 2000000) {
           if(mti_conex->espera_respuesta == mti_no) {
              punt_dato = mti_buscar_datoenviar(mti_conex->num_conex);
              if(punt_dato == NULL) {
                return 4;  /*como no se esperaba ninguna trama de respuestas,
                               no se retransmitira la trama anterior */
              } else {
                punt_dato->enviado = mti_no;
                mti_fin_procesoestados();
                return 1;
              }
           } else {
              /*se aumenta el contador de retransmisiones ya que se tiene
                 que volver a enviar el paquete */
              mti_conex->cont_retrans = mti_conex->cont_retrans + 1;
              if(mti_conex->cont_retrans >= mti_conex->temp) {
                return -1; /*devolvemos error porque ya hemos retransmitido el
                             dato mas de lo que devemos */
              } else {
                /*volvemos a enviar el dato que teniamos almacenado ya que
                   ha saltado el temporizador y aun no le ha llegado nada */
                MTI_BUF = &mti_trama_anterior;
                mti_convierte_formatoSCI(mti_datos_SCI);
                enviar_SCI1(Baud_115200, mti_datos_SCI, 41);
                if(mti_tam_dato >= 1) {
                   enviar_SCI1(Baud_115200, &MTI_BUF->mti_datos[0],
                            mti_tam_dato);
                }
              }
           }
        }
      }

      if(z == -1) {
        return -1;
      }
      /*el paquete no es para nosotros */
      if(z == 2) {
        return 2;
      }
      /*Si ha llegado un paquete bien y no es para nuestro puerto tenemos que
         almacenarlo en el buffer para que otra conexion la pueda utilizar.
         Buscamos que haya espacio para almacenarlo y si no sobreescribimos */
      if(MTI_BUF->puerto_dest != mti_conex->lc_puerto) {
        if(mti_BUFFER_PAQUETES[0].puerto_dest == 0) {
           /*meter MTI_BUF en mti_BUFFER_PAQUETES0 */
           mti_guardar_trama(0);
           return 3;
```

```
        } else if(mti_BUFFER_PAQUETES[1].puerto_dest == 0) {
           /*meter MTI_BUF en mti_BUFFER_PAQUETES1 */
           mti_guardar_trama(1);
           return 3;
        } else if(mti_BUFFER_PAQUETES[2].puerto_dest == 0) {
           /*meter MTI_BUF en mti_BUFFER_PAQUETES2 */
           mti_guardar_trama(2);
           return 3;
        } else {
           /*meter MTI_BUF en mti_BUFFER_PAQUETES0 */
           mti_guardar_trama(0);
           return 3;
        }
      }
   }


   /*Lo que aparece a continuacion es para obtener los datos que hay que
      pasarle al usuario */
   *tam_buffer = mti_tam_dato;
   if(mti_tam_dato > 0) {
      mti_conex->enviar_ack = mti_si;
      for(i = 0; i < mti_tam_dato; i++) {
        buffer[i] = MTI_BUF->mti_datos[i];
      }
   } else {
     /*si llega un pauqete y no lleva datos, no necesitara que envie respuesta */
      mti_conex->enviar_ack = mti_no;
      buffer[0] = 0;
      if(mti_conex->espera_respuesta == mti_no) {
        punt_dato = mti_buscar_datoenviar(mti_conex->num_conex);
        if(punt_dato == NULL) {
           return 4;       /*como no se esperaba ninguna trama de respuestas,
                             no se retransmitira la trama anterior */
        } else {
           punt_dato->enviado = mti_no;
        }
      }
   }
   /*se llama a la funcion que, segun lo que llegue y en el estado en el
      que se encuentre, evolucionara de estado y creara otro paquete */
   k = mti_proceso_estados();
   if(k <= 0) {
      return -1;
   }
   if(k == 5)
      return 5;
   return 1;
   }

/**************MTI_PROCESO_ESTADOS*********************************
Se procesa el estado en el que esta, la informacion que trae,
y a que estado evoluciona. Con toda la informacion se
crea la nueva trama.
*******************************************************************/

   int mti_proceso_estados() {
      int DATO;

      if(mti_conex->tcpestadobanderas != LISTEN) {

        /*Si lo que llega es un paquete reset ponemos el estado en cerrado y
        volvemos. */
        if(MTI_BUF->banderas & TCP_RST) {
           mti_conex->tcpestadobanderas = CLOSED;
           mti_borrar_estructura(mti_conex);
          /*borrar todos los nodo datos_a enviar que tengan este numero de
          conexion */
```

```
   return -1;
}

/*comprobamos si el numero de asentimiento es el que esperabamos, y si lo
es actualizamos */
if(MTI_BUF->ackno[0] == mti_conex->ack_nxt[0] &&
   MTI_BUF->ackno[1] == mti_conex->ack_nxt[1] &&
   MTI_BUF->ackno[2] == mti_conex->ack_nxt[2] &&
   MTI_BUF->ackno[3] == mti_conex->ack_nxt[3]) {
   mti_conex->snd_nxt[0] = mti_conex->ack_nxt[0];
   mti_conex->snd_nxt[1] = mti_conex->ack_nxt[1];
   mti_conex->snd_nxt[2] = mti_conex->ack_nxt[2];
   mti_conex->snd_nxt[3] = mti_conex->ack_nxt[3];
   DATO = mti_bien;
   mti_conex->cont_retrans = 0; /*Si el dato esta bien lo mando por
   primera vez */
} else {
   DATO = mti_mal; /*tendre que volver a reenviar el dato */
}
if(DATO == mti_mal) {
   mti_conex->cont_retrans = (mti_conex->cont_retrans) + 1;
   if(mti_conex->cont_retrans >= mti_conex->temp) {
      return -1;    /*devolvemos error porque ya hemos retransmitido el
                    dato mas de lo que devemos */
   } else {
      mti_fin_procesoestados(); /*volvemos a enviar el dato que enviamos
                                porque no ha llegado bien en numero de
                                secuencia que esperabamos */
      return 1;
   }
}

if(mti_conex->tcpestadobanderas == SYN_SENT) {
   /*En SYN_SENT, esperamos un SYNACK que es enviado en respuesta de
      nuestro SYN. El rcv_nxt es puesto para el numero de secuencia en el
      SYNACK mas uno, y nosotros enviamos un ACK. */

   if(MTI_BUF->banderas == (TCP_SYN | TCP_ACK)) {
      mti_conex->tcpestadobanderas = ESTABLISHED;
      mti_conex->rcv_nxt[0] = MTI_BUF->seqno[0];
      mti_conex->rcv_nxt[1] = MTI_BUF->seqno[1];
      mti_conex->rcv_nxt[2] = MTI_BUF->seqno[2];
      mti_conex->rcv_nxt[3] = MTI_BUF->seqno[3];
      mti_actualizar_rcv_nxt(1);
      /*aqui no deberiamos recibir datos, solo un SYNACK y enviar un ACK
      */
      mti_enviar_ack();
      return 1;
   }
   return -1;
}

/*chequeamos si el numero de secuencia que ha llegado es el que
   esperabamos. Si no, enviaresmos un ACK con el numero correcto */
if((MTI_BUF->seqno[0] != mti_conex->rcv_nxt[0] ||
    MTI_BUF->seqno[1] != mti_conex->rcv_nxt[1] ||
    MTI_BUF->seqno[2] != mti_conex->rcv_nxt[2] ||
    MTI_BUF->seqno[3] != mti_conex->rcv_nxt[3])) {
      mti_enviar_ack();   /*enviamos un ack para con el mismo num de seq
                          para que el otro extremo nos vuelva e enviar el
                          dato */
   return 5;
}

/*Hace diferentes cosas dependiendo en que estado este la conexion. */
switch (mti_conex->tcpestadobanderas & TS_MASK) {

 case SYN_RCVD:
```

```
      /*En SYN_RCVD hemos enviado un SYNACK en respuesta a un SYN, y
      esperarmos un ACK que asienta el dato que enviamos.entramos en el
      estado ESTABLISHED  y ya podemos enviar datos,no tenemos que borrar
      nigun nodo de datos a enviar porque es el primer dato que enviamos */

      if(MTI_BUF->banderas & TCP_ACK) {
        mti_conex->tcpestadobanderas = ESTABLISHED;
        /*buscamos si hay datos para enviar ya que la SYN_RCVD cuando
           llega el ACK ya puedo enviar datos */
        mti_fin_procesoestados(); /*ya puedo enviar datos */
        return 1;

      }
        /*si lo que llega no tiene un ack hay un error ya que la respuesta a
        un syn es un synack */
      return -1;

    case ESTABLISHED:

      /*Si el paquete que llega es un FIN, deberemos cerrar la conexion y
         enviar un FIN y entran en el estado LAST_ACK. Necesitamos que el
         FIN tenga el asentimiento de todos los datos, de otra manera
         abandonamos */

      /*si estamos aqui es porque el dato recibida esta bien. por tanto
         tenemos que borrar el nodo del dato que hemos recibido */
      mti_borrar_datoenviado(mti_conex->num_conex);

      if(MTI_BUF->banderas & TCP_FIN) {
        mti_actualizar_rcv_nxt(mti_tam_dato);
        mti_actualizar_ack_nxt(1);
        mti_conex->tcpestadobanderas = CLOSE_WAIT;
        mti_enviar_finack();
        return 1;
      }

      if(MTI_BUF->banderas & TCP_ACK) {
        mti_actualizar_rcv_nxt(mti_tam_dato);
        mti_fin_procesoestados();
        return 1;
      } else {
        return -1;
      }

    case CLOSE_WAIT:
      /*en este estado tenemos que enviar todos los datos pendientes
         y hasta que no esten todos enviados no pasara a LAST_ACK */
      if((MTI_BUF->banderas & TCP_FIN)
        || (MTI_BUF->banderas & TCP_ACK)) {
        mti_fin_procesoestados();
        return 1;
      }
      break;

    case LAST_ACK:
      /*podemos cerrar la conexion si el par ha asentido nuestro FIN. */

      if(MTI_BUF->banderas == (TCP_FIN | TCP_ACK)) {
        mti_conex->tcpestadobanderas = CLOSED;
        mti_borrar_estructura(mti_conex);
        return 1;
      }
      break;

    case FIN_WAIT_1:
      /*La aplicacion ha cerrado la conexion, pero el puerto remoto
         no ha la cerrado aun. Asi no hacemos nada pero esperamos por un
```

```
               fin desde el otro lado. */

            if(MTI_BUF->banderas == TCP_FIN) {
              mti_conex->tcpestadobanderas = CLOSING;
              mti_actualizar_rcv_nxt(1);
              mti_enviar_ack();
              return 1;
            } else if(MTI_BUF->banderas == (TCP_FIN | TCP_ACK)) {
              mti_conex->tcpestadobanderas = FIN_WAIT_2;
              mti_actualizar_rcv_nxt(1);
              mti_enviar_ack();
              return 1;
            }
            if(MTI_BUF->banderas & TCP_ACK) {
              mti_enviar_ack();
              return 1;
            }
            return -1;

         case FIN_WAIT_2:
            if(MTI_BUF->banderas & TCP_FIN) {
              mti_conex->tcpestadobanderas = TIME_WAIT;
              mti_conex->temp = 0;
              mti_actualizar_rcv_nxt(1);
              mti_enviar_finack();
              return 1;
            }
            if(MTI_BUF->banderas & TCP_ACK) {
              mti_enviar_ack();
              return 1;
            }
            return -1;

         case TIME_WAIT:
            /*tendremos que esperar un tiempo para asegurar que le llega el ack
               al otro extremo */
            mti_conex->temporizador2 = 0;
            while(mti_conex->temporizador2 <= 100) { /*equivale  a  0,125  segundos
                                                       */
            };                   /*no hacemos nada hasta que no pase este tiempo
                                   */

            mti_conex->tcpestadobanderas = CLOSED;
            mti_borrar_estructura(mti_conex);
            return 1;

         case CLOSING:
            if(MTI_BUF->banderas & TCP_FIN) {
              mti_conex->tcpestadobanderas = TIME_WAIT;
              mti_conex->temp = 0;
              return 1;
            }
            if(MTI_BUF->banderas & TCP_ACK) {
              mti_enviar_ack();
            }
        }
      }
      return 1;
   }
/********MTI_FIN_PROCESOESTADOS************************
saltamos si hay posibles datos para enviar
**********************************************/

   void mti_fin_procesoestados() {
      int j = 0;
      SEND *punt_dato;

      punt_dato = mti_buscar_datoenviar(mti_conex->num_conex);
```

```
      if(punt_dato == NULL) {    /*no hay datos para enviar a esta conexion */
        if(mti_conex->enviar_fin == mti_si) {
           mti_conex->enviar_fin = mti_no;
           if(mti_conex->tcpestadobanderas == CLOSE_WAIT) {
              mti_conex->tcpestadobanderas = LAST_ACK;
           } else {
              mti_conex->tcpestadobanderas = FIN_WAIT_1;
           }
           mti_enviar_fin();
           return;
        } else if(mti_conex->tcpestadobanderas == CLOSE_WAIT) {
           /*si estamos en close_wait y no hay datos que enviar no manda
              nada hasta que no cierren la conexion */
           mti_conex->espera_close = mti_si;
           return;
        }
        if(mti_conex->enviar_ack == mti_si) {
           mti_enviar_ack();      /*enviamos un ack */
           return;
        } else {
           mti_conex->espera_respuesta = mti_no;
           return;
        }
      } else {
        mti_conex->espera_respuesta = mti_si;
        mti_tam_dato = punt_dato->tam;
        for(j = 0; j < mti_tam_dato; j++) {
           MTI_BUF->mti_datos[j] = punt_dato->info[j];
           /*Cambio esto *((punt_dato->info)++) por punt_dato->info[j] */
        }

        mti_borrar_datoenviado(mti_conex->num_conex);
        mti_actualizar_ack_nxt(mti_tam_dato);
        MTI_BUF->banderas = TCP_ACK;
        mti_enviar_trama();
        return;
      }
   }
```

- ***mti_trama_completa_rx.c***

```
#include "mti.h"
#include <stdio.h>

/******************MTI_TAMA_COMPLETA*****************************************
Buscamos una trama completa.Si hay error devuelve -1. Si esta bien dara 1
y si aun no ha llegado la trama completa devuelve 0
*****************************************************************************/
int
mti_trama_completa_rx(int *cont_trama)
{
   int mti_espera_PPP(void);
   static int i, j;
   int k;
   INT16U aux1;
   INT16U aux2;
   INT16U cksum = 0;
   INT16U cksum_aux = 0;
   INT16U cksum_tcp = 0;

   if(first1 == (last1 - 1)) {
      return 0;
   }

   switch (*cont_trama) {
    case (0):
```

```
      k = mti_espera_PPP();
      if(k == 0) {
        return 0;
      } else {
        if(k == 1) {
           *cont_trama = *cont_trama + 1;
           return 0;
        } else {
           return -1;
        }
      }


   case (1):
      /*ESPERAMOS QUE LLEGUE TODA LA CABECERA )
         first1=40,last1=1 40-1=39 esperamos mientras sea meor que 39 o
         lo que es lo mismo, menor o igual que 38 */
      if((first1 - last1) <= 38) {
        return 0;
      } else {
        MTI_BUF->vhl = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->tipo = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->tam[0] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->tam[1] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        mti_tam_dato = (((MTI_BUF->tam[0] << 8) + MTI_BUF->tam[1]) - 40);
        MTI_BUF->ipid[0] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->ipid[1] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->ipoffset[0] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->ipoffset[1] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->ttl = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->proto = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        i = last1++;
        last1 = last1 % mti_TAM_BUFFER1;
        aux1 = ((buffer1[i] << 8) & 0xff00);
        aux2 = (buffer1[last1++] & 0xff);
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->ipchksum = aux1 + aux2;
        MTI_BUF->fuente_ip_dir[0] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->fuente_ip_dir[1] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->fuente_ip_dir[2] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->fuente_ip_dir[3] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->dest_ip_dir[0] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->dest_ip_dir[1] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;
        MTI_BUF->dest_ip_dir[2] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;

        MTI_BUF->dest_ip_dir[3] = buffer1[last1++];
        last1 = last1 % mti_TAM_BUFFER1;

        i = last1++;
        last1 = last1 % mti_TAM_BUFFER1;
        aux1 = ((buffer1[i] << 8) & 0xff00);
```

```
      aux2 = (buffer1[last1++] & 0xff);
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->puerto_fuente = aux1 + aux2;
      j = last1++;
      last1 = last1 % mti_TAM_BUFFER1;
      aux1 = ((buffer1[j] << 8) & 0xff00);
      aux2 = (buffer1[last1++] & 0xff);
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->puerto_dest = aux1 + aux2;
      MTI_BUF->seqno[0] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->seqno[1] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->seqno[2] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->seqno[3] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->ackno[0] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->ackno[1] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->ackno[2] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->ackno[3] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->tcpoffset = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->banderas = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->wnd[0] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->wnd[1] = buffer1[last1++];
      last1 = last1 % mti_TAM_BUFFER1;

      i = last1++;
      last1 = last1 % mti_TAM_BUFFER1;
      aux1 = ((buffer1[i] << 8) & 0xff00);
      aux2 = (buffer1[last1++] & 0xff);
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->tcpchksum = aux1 + aux2;
      i = last1++;
      last1 = last1 % mti_TAM_BUFFER1;
      aux1 = ((buffer1[i] << 8) & 0xff00);
      aux2 = (buffer1[last1++] & 0xff);
      last1 = last1 % mti_TAM_BUFFER1;
      MTI_BUF->urgp = aux1 + aux2;
      if(mti_tam_dato == 0) {
         break;
      } else {
         *cont_trama = *cont_trama + 1;
         return 0;
      }
     }

  case (2):
     /*esperamos a que nos hayan llegado todos los datos */
     if((first1 - last1) < (mti_tam_dato - 1)) {
       return 0;
     } else {
       for(i = 0; i < mti_tam_dato; i++) {
          MTI_BUF->mti_datos[i] = buffer1[last1++];
          last1 = last1 % mti_TAM_BUFFER1;
       }
     }
     break;
 }

 /*FIN DEL SWITCH, YA TENEMOS TODA LA ESTRUCTURA ACTUALIZADA
```

```
      PASAMOS A LAS COMPROBACIONES */


   /*Comprovamos la version */
   if(MTI_BUF->vhl != 0x45) {
      return -1;
   }

   /*Comprobamos si la direccion ip es la nuestra, es decir,
      si la información es para nosotros. */
   /*Si devuelve 2 es que el paquete no es para nosotros */
   if(MTI_BUF->dest_ip_dir[0] != MTI_IP_DIR0) {
      return 2;
   }
   if(MTI_BUF->dest_ip_dir[1] != MTI_IP_DIR1) {
      return 2;
   }
   if(MTI_BUF->dest_ip_dir[2] != MTI_IP_DIR2) {
      return 2;
   }
   if(MTI_BUF->dest_ip_dir[3] != MTI_IP_DIR3) {
      return 2;
   }

   /*      comprobamos el checksum IP */
   cksum_aux = MTI_BUF->ipchksum;
   MTI_BUF->ipchksum = 0;
   cksum = mti_ipchksum();
   MTI_BUF->ipchksum = cksum_aux;
   if(MTI_BUF->ipchksum != cksum) {
      return -1;
   }

   /*Comprovamos si el protocolo es TCP */
   if(MTI_BUF->proto != 0x06) {
      return -1;
   }

   /*      comprobamos el checksum ip */
   cksum_aux = MTI_BUF->tcpchksum;
   MTI_BUF->tcpchksum = 0;
   cksum_tcp = mti_tcpchksum();
   MTI_BUF->tcpchksum = cksum_aux;
   if(MTI_BUF->tcpchksum != cksum_tcp) {
      return -1;
   }
   return 1;
}

/*****************MTI_ESPERA_PPP*****************************
Espera a que le llegue la cabecera PPP
Aqui se comprobara si la cebecera PPP llega bien o tiene error
Si hubiera error devolveria -1;
************************************************************/
int
mti_espera_PPP(void)
{
   if(buffer1[last1++] != 0x11) {
      last1 = last1 % mti_TAM_BUFFER1;
      return 0;
   } else {
      return 1;
   }
}
```

- ***mti_write.c***

```
#include "mti.h"
#include <stdio.h>

#include "mti.h"
//#include <stdio.h>

/*******************MTI_WRITE**************************
Almacena un dato en una estructura
******************************************************/
int
mti_write(int socket, INT8U * mensaje, INT16U tam_mensaje)
{
   SEND *paux;
   SEND *auxiliar;
   MTI_ESTRUCTURA_CONEX *puntero;
   int i;

   puntero = mti_buscar_id_conex(socket);
   if(puntero == NULL) {
      return -1;    /*no existe la conexion para la que queremos escribir */
   }

   if(puntero->enviar_fin == mti_si) {  /*como  se  ha  llamado  a  close  no  debo
                                          enviar mas datos */
      return -1;
   }

   if(puntero->tcpestadobanderas == FIN_WAIT_1) {
      /*error, la conexion se esta cerrando */
      return -1;
   }

   if(puntero->tcpestadobanderas == FIN_WAIT_2) {
      /*error, la conexion se esta cerrando */
      return -1;
   }

   if(puntero->tcpestadobanderas == CLOSING) {
      /*error, la conexion se esta cerrando */
      return -1;
   }

   if(puntero->tcpestadobanderas == LAST_ACK) {
      /*error, la conexion se esta cerrando */
      return -1;
   }

   if(puntero->tcpestadobanderas == TIME_WAIT) {
      /*error, la conexion se esta cerrando */
      return -1;
   }

   if(puntero->tcpestadobanderas == LISTEN) {
      /*error, aun no existe esa conexion */
      return -1;
   }

   /*Caso 1:aun no existe ningun dato */
   if(mti_datos_a_enviar == NULL) {
      /*reservamos memoria para esta estructura. */
      mti_datos_a_enviar = &mti_datos_a_enviar1;
      paux = mti_datos_a_enviar;
      paux->num_conex = socket;
      paux->tam = tam_mensaje;
```

```
      paux->enviado = mti_no;
      for(i = 0; i < tam_mensaje; i++) {
        paux->info[i] = *(mensaje++);
      };
      paux->proximo = NULL;
      return 1;
   } else {
      paux = mti_datos_a_enviar;
      while(paux != NULL) {
        auxiliar = paux;
        paux = paux->proximo;
      };

      /*reservamos memoria para esta estructura. */
      auxiliar->proximo = mti_malloc_enviar();
      if(auxiliar->proximo == NULL) {
        return -1;
      } else {
        paux = auxiliar->proximo;
        paux->num_conex = socket;
        paux->tam = tam_mensaje;
        for(i = 0; i < tam_mensaje; i++) {
           paux->info[i] = *(mensaje++);
        };
        paux->proximo = NULL;
        return 1;
      }
   }
}

/*************MTI_BUSCAR_DATOSENVIAR************************
Busca datos que pertenezcan a una conexion para enviarlos
*************************************************/

SEND *
mti_buscar_datoenviar(int x)
{
   SEND *paux;
   int encontrado = 0;

   paux = mti_datos_a_enviar;
   while((paux != NULL) && (!encontrado)) {
      if(paux->num_conex == x) {
        paux->enviado = mti_si;
        encontrado = 1;
      } else {
        paux = paux->proximo;
      }
   }
   return paux;
}


/************MTI_BORRAR_DATOENVIADO*****************************
borra la primera estructura que encontremos con un determinado
identificador porque ya no necesitamos esa informacion
*************************************************************/
void
mti_borrar_datoenviado(int x)
{
   SEND *paux;
   SEND *pant;

   if(mti_datos_a_enviar == NULL)
      return;
   paux = mti_datos_a_enviar;
   if(paux->num_conex == x) {   /*borramos el primer elemento de la estructura */
      if(paux->enviado == mti_no) {
```

```
        return;
      } else {
        if(paux->proximo != NULL) {
           mti_datos_a_enviar = paux->proximo;
        } else {
           mti_datos_a_enviar = NULL;   /*no hay ninguna dato mas */
        }
        mti_free_enviar(paux);
        return;
      }
   } else {               /*no es la primera estructura la que borramos */
      pant = paux;
      paux = paux->proximo;      /*como paux apunta a la primera estructura y ya
                                 hemos comprobado que esta no es la que
                                 borramos,actualizamos paux */
      if(paux->num_conex == x) {
        if(paux->enviado == mti_si) {
           pant->proximo = paux->proximo;
           mti_free_enviar(paux);
        }
      } else {
        pant = paux;
        paux = paux->proximo;
      }
   }
   return;
}
```

- ***otros.c***

```
#include "mti.h"
#include <stdio.h>

/************MTI_BUSCAR_TCPESTADOBANDERAS*****************
Busca el estado de un nodo
*****************************************************/

MTI_ESTRUCTURA_CONEX *
mti_buscar_tcpestadobanderas(MTI_ESTRUCTURA_CONEX * conexion, int x)
{
   MTI_ESTRUCTURA_CONEX *paux;
   int encontrado = 0;

   paux = conexion;
   while((paux != NULL) && (!encontrado)) {
      if(paux->tcpestadobanderas == x) {
        encontrado = 1;
      } else {
        paux = paux->proximo;
      }
   };
   return paux;
}

/******************MTI_BUSCAR_CONEX_LIBRE**********************
Busca una conexion que no este siendo utilizada
*****************************************************/

int
mti_buscar_conex_libre(void)
{
   int j = 0;

   for(j = 0; j < mti_num_max_conex; ++j) {
      if(mti_num_conexion[j] == mti_libre) {
        mti_num_conexion[j] = mti_ocupado;
```

```
        return (j + 1);
      }
   };
   return -1;
}


/******MTI_BUSCAR_ID_CONEX***********************************
Busca una conexion cuyo identificador es el que se le
pasa como parametro
*****************************************************/
MTI_ESTRUCTURA_CONEX *
mti_buscar_id_conex(int x)
{
   MTI_ESTRUCTURA_CONEX *paux;
   int encontrado = 0;

   paux = mti_conexion;
   while((paux != NULL) && (!encontrado)) {
      if(paux->num_conex == x) {
        encontrado = 1;
      } else {
        paux = paux->proximo;
      }
   }
   return paux;
}

/******MTI_BUSCAR_PUERTO**************************
Busca un nodo cuyo puerto sea el que se le pasa
como parametro
*****************************************/

MTI_ESTRUCTURA_CONEX *
mti_buscar_puerto(MTI_ESTRUCTURA_CONEX * conexion, INT16U x)
{
   MTI_ESTRUCTURA_CONEX *paux;
   int encontrado = 0;

   paux = conexion;

   while((paux != NULL) && (!encontrado)) {
      if(paux->lc_puerto == x)
        encontrado = 1;
      else
        paux = paux->proximo;
   }
   return paux;
}

/*****************MTI_BORRAR_ESTRUCTURA*************************
Borra una estructura
*****************************************************/

void
mti_borrar_estructura(MTI_ESTRUCTURA_CONEX * conex)
{
   MTI_ESTRUCTURA_CONEX *paux;
   MTI_ESTRUCTURA_CONEX *pant;
   int x;

   x = conex->num_conex;
   mti_num_conexion[x] = mti_libre;
   paux = mti_conexion;
   if(paux->num_conex == x) {   /*borramos el primer elemento de la estructura */
      if(paux->proximo != NULL) {
        mti_conexion = paux->proximo;
      } else {
```

```
        mti_conexion = NULL;      /*no hay ninguna conexion mas */
      }
      mti_free_conex(paux);
      return;
   } else {               /*no es la primera estructura la que borramos */
      pant = paux;
      paux = paux->proximo;      /*como paux apunta a la primera estructura y ya
                                 hemos comprobado que esta no es la que
                                 borramos,actualizamos paux */
      if(paux->num_conex == x) {
        pant->proximo = paux->proximo;
        mti_free_conex(paux);
        mti_num_conexion[x] = mti_libre;
      } else {
        pant = paux;
        paux = paux->proximo;
      }
   }
   return;
}
```