UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ingeniería

Implantación del código convolucional con decodificación de Viterbi para la detección de errores en un sistema de comunicación inalámbrica

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

INGENIERO ELÉCTRICO-ELECTRÓNICO

PRESENTA:

ROGELIO ESTEBAN ARVEA PÉREZ

DIRECTOR DE TESIS:

Dr. Carlos Rivera Rivera

MÉXICO,D.F. Marzo 2004

Agradecimientos

Se ha llegado al final de un ciclo; una de las principales metas en la vida se ha alcanzado. Es el momento de volver la vista atrás y de ver qué tan bien se ha llegado, de revisar qué es lo que se ha hecho mal y corregirlo, de ver que es lo que se ha hecho bien y superarlo. Pero también es el momento de algo muy importante, es el momento de dar gracias:

Gracias a Dios por prestarme vida y permitirme llegar hasta este momento, gracias a Dios por darme a mis padres que siempre han estado como parte primordial en el transcurrir de mi vida, gracias a Dios por permitirme conocer a tanta y tanta gente que han puesto, sin quererlo o no, parte de su persona para permitirme ser lo que ahora soy.

A mis padres. No hay palabras para agradecer el sacrificio y el trabajo que diario a diario, día a día y siempre han hecho para hacer todo lo que hasta ahora, poco o mucho, he podido construir, solo espero no haberlos defraudado y haber correspondido a toda la confianza que ustedes han depositado en mi. Gracias por que con su incondicional apoyo y su constante compañía me han permitido superar los momentos más difíciles en mi vida. No tengo más que decir, solo que mi vida es de ustedes, gracias por dármela.

A mis profesores, que con sus enseñanzas me han formado para hacer frente a una vida cada vez más difícil. Gracias por darme más que teoremas y teorías, gracias por darme experiencias propias de sus vidas. Quizá seré un reflejo de ustedes, gran parte de ustedes ha quedado en mi

A los cuates, a los compañeros de parranda, José Vizcaya alias el "vizcayas", Vicente Reyes alias "chente", Ismael Reyes, Leopoldo Islas alias "polo", David, Armando alias "Armandus", el "filetes", el "chayote", Miguel Hidalgo alias el "tío nieblas", Oscar etc. que tantas y tantas aventuras hemos pasado, con ustedes compartí uno de los periodos más hermosos de la vida, la vida universitaria, con ustedes viví tantas cosas tan tremendas, que a la distancia no queda más que reír y guardarlas para el anecdotario que algún contaré a mis nietos. Gracias por su compañía. Y de manera especial, tal y como lo prometí, dedico al "vizcayas" los capítulos 1 y 2 por su apoyo en momento muy difíciles para mí, gracias por tu apoyo.

A la mujer que amo, porque con ella he sabido lo que es tener una verdadera mujer a mi lado. Gracias Jeny, por que me siento afortunado de que me permitas ser parte de tu vida, espero que sean muchos y muchos años los que permanezcamos juntos, nos falta mucho por vivir, muchas cosas por las que hay que pasar, pero estando a tu lado se que todo será más bello. Te amo.

Un agradecimiento especial a los profesores Carlos Rivera Rivera, director de esta tesis y Fernando Lepe profesor de Postgrado, que han colaborado de manera importante en la realización de esta tesis. Sin ustedes esto no hubiera sido posible.

ÍNDICE

INTRODUCCIÓN

Objetivo

El propósito de este trabajo es introducirnos hacia la técnica de corrección de errores conocida como codificación convolucional con decodificación de Viterbi. Más particularmente, se enfoca primeramente al algoritmo de decodificación de Viterbi solamente. Luego intenta despertar el interés hacia el diseño o el entendimiento de los sistemas de comunicación digital inalámbricos.

Definición del problema

Hoy en día no es sorprendente que la información se codifique con objeto de controlar (detectar y corregir) los errores que se originan como consecuencia del paso de las señales por el medio de transmisión. Es más, resulta difícil pensar en un sistema que no incorpore tales mecanismos. Si se analizara el lenguaje natural, se observaría en que existen multitud de reglas (fonéticas, ortográficas, léxicas, sintácticas,...) que permiten que sólo muy pocos de los posibles mensajes recibidos tengan sentido. Debido a esta redundancia se puede distinguir un mensaje que llega "limpio" al receptor de otro que se ha visto distorsionado por el medio a través del cual ha sido transmitido.

Por otra parte, en caso de recibir un mensaje erróneo no es extraño que el subsistema receptor trate de buscar el mensaje (con sentido) que se asemeje más al recibido, es decir, que cumpla las reglas que permiten admitir el mensaje recibido como "bueno" con el mínimo cambio significativo sobre el mismo. Tampoco resulta nada raro que el receptor del mensaje pida la retransmisión de aquel en caso de verse incapaz de reconstruir la información desvirtuada.

El precio que se paga por disponer de un subsistema capaz de reducir la probabilidad de recibir un mensaje erróneo es aumentar la longitud del mensaje. Así pues, la importancia del sistema corrector de errores estriba en disminuir los errores en recepción, pero aumentando la longitud del mensaje de forma mínima, o bien intercambiando tiempo de transmisión por fidelidad de la información recibida.

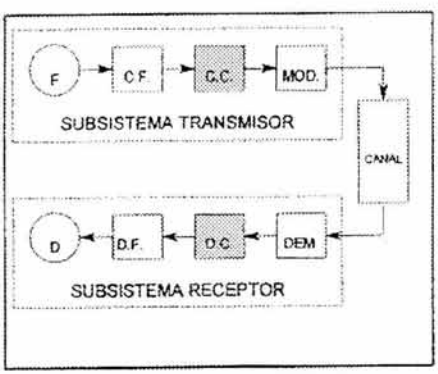


Figura A Sistema de comunicación simple

La protección de un sistema contra las perturbaciones que puedan afectar a la información transmitida se lleva a cabo en el transmisor (a la entrada del modulador). De forma análoga, la eliminación de toda la redundancia que introduce dicha protección se efectúa en el receptor (a la salida del demodulador), tal y como se puede observar en la figura A.

La codificación de canal desarrolla y analiza códigos orientados a facilitar la detección y corrección de los errores ocasionales que pueda introducir el canal de transmisión. En todo sistema de transmisión de la información es tan importante disponer de una buena codificación de canal, como lo puedan ser la integridad y la fiabilidad de la información que alcanza al destinatario.

Además, con lo que respecta al sistema transmisor y receptor, para establecer el mecanismo que se usará en la transmisión de datos a través del puerto serie de la PC, debemos tomar en cuenta dos posibles métodos, el método de una revisión constante del puerto para verificar si existe un mensaje recibido o el mecanismo de interrupcion del procesador, en el cual se utiliza una prestación de la arquitectura de la PC que consiste en avisar al procesador de la llegada de un mensaje en el puerto serie, en el momento que éste llega, para darle el manejo que se requiera. Además, como parte de la implementación del sistema de transmisión y recepción, debemos tener presente el uso algún dispositivo que facilite el establecimiento del enlace inalámbrico bidirecciónal y que se adapte a las especificación de tipo eléctrico que caracterizan al puerto serie

Después de la introducción, se provee una detallada descripción de los algoritmos para la generación de datos binarios aleatorios, codificación convolucional de los datos, pasando los datos codificados a través de un canal con ruido, cuantizando los símbolos recibidos de canal, y desarrollando la decodificación de Viterbi sobre los símbolos cuantizados del canal para recobrar los datos binarios originales. Los ejemplos de código fuente para simulación de estos algoritmos siguen a estas descripciones. La descripción de cómo se implementó la interfáz de usuario y el mecanismo de manejo de mensajes recibidos y transmitidos son tratados en capítulos aparte. Finalmente, se toca el tema de los dispositivos utilizados para el establecimiento del enlace. También se incluyen algunos resultados de la simulación del código

Antecedentes

La codificación convolucional con decodificación de Viterbi es una técnica de Corrección de errores conforme se recibe el mensaje, FEC (Forward Error Correction) cuyo sentido es tratar de detectar y corregir errores en un mensaje tal y como se va recibiendo, en el receptor, hacia adelante, sin necesidad de pedir una retransmisión, y es particularmente dirigido a un canal en el cual la señal transmitida es corrompida principalmente por ruido gaussiano blanco aditivo (Additive White Gaussian Noise AWGN). Se puede pensar en el AWGN como aquel ruido cuya distribución de voltaje sobre el tiempo tiene una característica que puede ser descrita usando distribución

Gaussiana o normal, esto es, una curva de campana. Esta distribución de voltaje tiene una media cero y una desviación estándar que es una función de la relación señal a ruido (SNR) de la señal recibida. Asuma por el momento que el nivel de la señal recibida es fija. Entonces si la SNR es alta, la desviación estándar del ruido es pequeña, y viceversa. En comunicaciones digitales, la SNR es usualmente medida en términos de E_b/N_0, el cual muestra la energía por bit dividida por la densidad de ruido unilateral.

La decodificación de Viterbi fue desarrollada por Andrew J. Viterbi, fundador de Qualcomm Corporation. Su documento sobre la técnica de decodificación es " Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm ", publicado en IEEE Transactions on Information Theory, Volumen IT-13, páginas 260-269, en abril de 1967. Desde entonces, otros investigadores han ampliado su trabajo encontrando buenos códigos convolucionales, explorando los límites del funcionamiento de la técnica, y variando parámetros de diseño del decodificador para optimizar la puesta en práctica de la técnica en hardware y software. El algoritmo decodificador de Viterbi también se utiliza en la decodificación de la modulación codificada en enrejado (trellis-coded modulation TCM), la técnica usada en módems de línea telefónica para exprimir altos cocientes de bits-por-segundo a Hertz fuera de los 3 kHZ de ancho de banda de las líneas telefónicas analógicas.

El algoritmo de Viterbi esencialmente desarrolla la decodificación de máxima probabilidad; además, este reduce la carga de cómputo tomando ventaja de la estructura especial en el código trellis. La ventaja de la decodificación de Viterbi, comparada con la decodificación con fuerza bruta, es que la complejidad de un decodificador de Viterbi no esta en función del número de símbolos en la secuencia de la palabra codificada. El algoritmo involucra el cálculo de una medida de similitud o distancia, entre la señal recibida, en un tiempo t, y todos los caminos del enrejado que entran a cada estado en un tiempo t. El algoritmo de Viterbi quita, con base en ciertas consideraciones, aquellos caminos que no pueden escogerse como posibles candidatos para la máxima probabilidad. Cuando dos caminos entran al mismo estado, se escoje aquel que tiene la mejor métrica; este camino se le llama el camino superviviente. Esta selección de caminos supervivientes es desarrollada para todos los estados. El decodificador continúa de esta forma hasta profundizar en el diagrama de trellis, haciendo decisiones para ir eliminando los caminos menos probables. El rechazo temprano de los caminos menos probables reduce la complejidad de la decodificación. En 1969, Omura demostró que el algoritmo de Viterbi es, de hecho, el que desarrolla la máxima probabilidad. Note que la meta de seleccionar el camino óptimo puede expresarse, equivalentemente, como escoger la palabra codificada con la máxima medida de probabilidad, o como escoger la palabra codificada con la mínima medida de distancia.

Capítulo 1

Codificación de Canal

Capítulo 1. Codificación de canal

El propósito de la Corrección de errores conforme se recibe el mensaje (Forward Error Correction FEC) es mejorar la capacidad de un canal por medio de agregar información redundante cuidadosamente diseñada para que los datos sean transmitidos a través del canal. Al proceso de agregar esta información redundante se le coonoce como codificación del canal. La codificación convolucional y la codificación de bloque son las dos mayores formas de la codificación del canal. Los códigos convolucionales operan en datos seriales, en uno a en algunos bits en un tiempo. Los códigos de bloque operan en bloques de mensaje relativamente largos (típicamente, arriba de un par de cientos de bytes). Hay una variedad de códigos convolucionales y de bloque usados comúnmente, y una variedad de algoritmos para decodificar las secuencias de información codificada recibida para recuperar los datos originales. Un código lineal de bloque es descrito por dos números enteros, n y k, y una matriz o polinomio generadores. El entero k es el número de bits de datos que forman una entrada al codificador de bloque. El entero n es el número toral de bits en el mensaje codificado en la salida del codificador. Una característica de los códigos lineales de bloque es que cada símbolo de n bits codificado es únicamente determinado por el mensaje de entrada de k bits. Se le llama a la razón k/n la tasa o razón de código – una medida de la cantidad de redundancia sumada. Un código convolucional es descrito por tres números enteros n, k, y K, donde la razón k/n tiene el mismo significado de razón de código (información por bit codificado, se expresa como el cociente del número de bits en el codificador covolucional (k) al número de símbolos de canal que salen por el codificador convolucional (n) en un ciclo dado del codificador) que tiene para los códigos de bloque. El entero K es un parámetro conocido como la longitud restringida o en inglés *constraint length*; denota la "longitud" del codificador convolucional, es decir cuántas etapas de K-bits están disponibles para alimentar la lógica combinatoria que produce los símbolos de la salida esto representa el número de etapas en el registro de desplazamiento codificador. Una importante característica de los códigos convulacionales, a diferencia de los códigos de bloque, es que el codificador tiene memoria , la n-upla emitida por el procedimiento de codificación convolucional es no solamente una función de la k-upla de entrada, sino que también es una función de la (K-1) k-upla de entrada. En la practica, n y k son números enteros pequeños y K es variada para el control de la redundancia. Entonces, se relaciona cercanamente con K, el parámetro m, que indica cuántos ciclos del codificador un bit se conserva y se utiliza para la decodificación después de que este aparece en la entrada del codificador convolucional. El parámetro m se puede pensar como la longitud de memoria del codificador.

1.1 Codificación Convolucional

Una versión de un diagrama de bloques de un sistema digital de comunicaciones, haciendo énfasis en las porciones del codificador/decodificador convolucional y el modulador/demodulador del enlace de comunicaciones, es la figura 1.1. La fuente del mensaje de entrada es denotado por la secuencia $\mathbf{m} = m_1, m_2, \ldots, m_i, \ldots$ donde cada mi representa un dígito binario (bit). Donde asumimos que cada m_i es igualmente probable de ser un uno o un cero, y además es independiente de los otros dígitos. Siendo independiente,

la secuencia de bits pierde cualquier redundancia; esto es, el conocimiento acerca del bit m_i, no da información acerca de m_j ($i \neq j$). El codificador transforma cada secuencia m en una secuencia única codificada U = G(m). Aunque a través de la secuencia m definimos una única secuencia U, una característica principal de los códigos convolucionales es que dada una k-upla en m no solo define una única n-upla asociada en U debido a que la codificación de cada k-upla no es solo una función de aquella k-upla sino que también es una función del K-1 k-uplas que la preceden. La secuencia U puede ser partida en una secuencia de ramas de palabras: $U= U_1,U_2, \ldots, U_i, \ldots$. Cada palabra de rama U_i es hecha de un símbolo codificado binario, llamados comúnmente símbolos de canal, bits de canal, o bits codificados; a diferencia del mensaje de bits en la entrada, los símbolos codificados no son independientes.

En una aplicación típica de comunicaciones, la secuencia de símbolos codificados U modulan una forma de onda s(t). Durante la transmisión, la forma de onda s(t) es corrompida por ruido, resultando en una forma de onda recibida s´(t) y una secuencia demodulada $Z= Z_1, Z_2, \ldots, Z_i, \ldots$ como se indica en la figura 1.1. La misión del decodificador es producir una estimado $m' = m'_1, m'_2, \ldots, m'_i, \ldots$ de la secuencia original del mensaje, usando la secuencia recibida Z con un conocimiento del procedimiento de codificación .

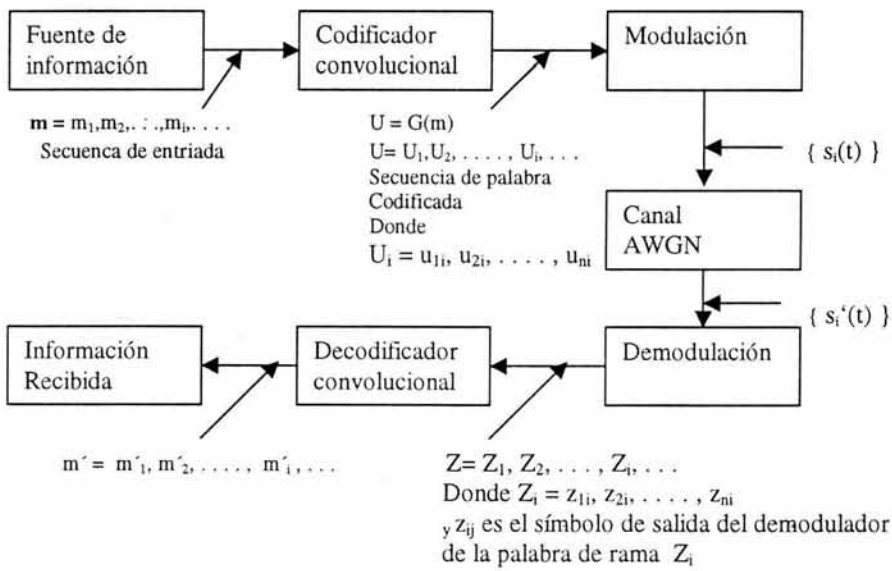


Figura 1.1 Porciones de codificación/decodificación y modulación/demodulación de un enlace de comunicaciones

Un codificador convolucional general, mostrado en la figura 1.2, se representa con un registro de desplazamiento de kK-etapas y n sumadores modulo-2, donde K es la longitud restringida.

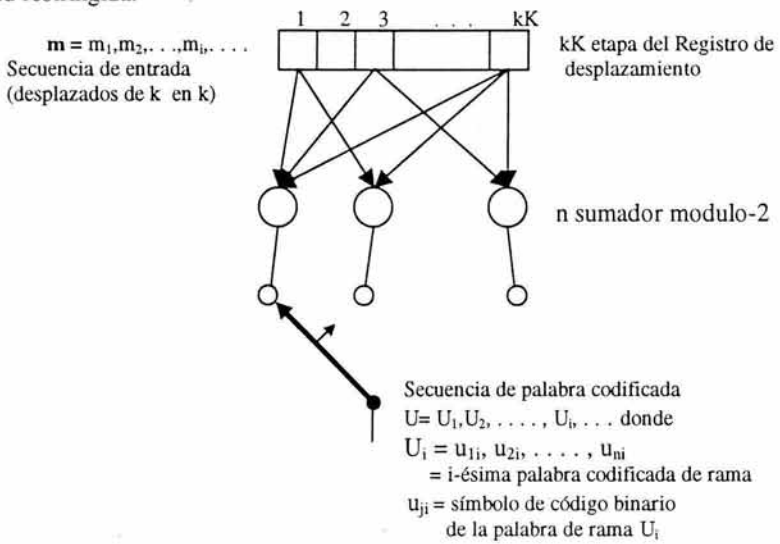


Figura 1.2 Codificador convolucional con longitud restringida K y razón k/n

La longitud restringida representa el número de k-bits desplazados, sobre los cuales un solo bit de información puede influir en la salida del codificador. En cada unidad de tiempo, k bits son desplazados dentro de las primeras k etapas del registro; todos los bits en el registro son desplazados k etapas a la derecha, y las salidas de los n sumadores son secuencialmente muestreados para formar los símbolos codificados binarios o los bits codificados. Estos símbolos codificados son entonces usados por el modulador para dar la forma de onda que va a ser transmitida sobre el canal. Debido a que hay n bits codificados por cada grupo de entrada de k bits de mensaje, la tasa de código es k/n de bits de mensaje por bit de código, donde k<n.

Debemos considerar solo el codificador convolucional binario más común, para el cual k = 1, que es de aquellos codificadores en los cuales los bits del mensaje son desplazados dentro del codificador un bit a la vez. Para el codificador k = 1, en la unidad i de tiempo, el bit de mensaje mi, es desplazado dentro del la primera etapa del registro; todos los bits previos en el registro son desplazados una etapa a la derecha, y como en el caso más general, las salidas de los n sumadores son secuencialmente muestreadas y transmitidas. Debido a que hay n bits de código por cada bit de mensaje, la razón de código es 1/n. Los n símbolos0 de código ocurridos en el tiempo t_i comprenden la i-ésima palabra $U_i = u_{1i}, u_{2i}, \ldots, u_{ni}$, donde u_{ji} (j = 1, 2 . . . , n) es el j-ésimo símbolo codificado perteneciendo a la i-ésima palabra. Note que para el codificador de tasa 1/n, el registro de desplazamiento de kK-etapas puede ahora ser referido simplemente como un registro de K-etapas, y la longitud restringida K, el cual fue expresado en unidades de etapas k-uplas, puede ahora ser referido como la longitud restringida en unidades de bits.

1.2 Representación del Codificador Convolucional.

Para describir un código convolucional, necesitamos caracterizar la función de codificación G(**m**), de tal manera que dada una secuencia de entrada **m**, podamos ser capaces de calcular la secuencia de salida **U**. Muchos métodos son usados para representar un codificador convolucional, los más populares son conexión de vectores o polinomios, el diagrama de estados, el diagrama de árbol y el diagrama de trellis(enrejado). A cotinuación descrebiremos algunos de ellos pero centraremos nuestra atención en el diagrama de trellis.

1.2.1 Representación de conexiones

Usaremos el codificador convolucional mostrado en la figura 1.3 como un modelo para la discusión de los codificadores convolucionales. La figura ilustra un codificador convolucional (2,1) con una longitud restringida k= 3. Hay n = 2 sumadores modulo 2; entonces la razón de código k/n es ½. En cada tiempo de bit de entrada, un bit es desplazado dentro de la etapa mas a la izquierda y los bits dentro del registro son desplazados una posición hacia la derecha. Luego, el dispositivo de la salida muestrea la salida de cada sumador modulo-2 (para este caso el sumador de arriba y el sumador de abajo), entonces formando el par de símbolos de código estamos haciendo la palabra de rama asociado con el bit que hemos acabamos de introducir. El muestreo es repetido para cada bit que introduzcamos. La selección de la conexión entre los sumadores y las etapas del registro nos da la característica del código. Cualquier cambio en la elección de las conexiones resulta en un diferente código. Las conexiones son, por supuesto, no escogidas o cambiadas arbitrariamente. El problema de escoger las conexiones para obtener una buena propiedad de distancia es complicada y no ha sido resuelta en general; Sin embargo, códigos buenos han sido encontrados por búsqueda de computadores para todas las longitudes restringidas menores que 20.

A diferencia de un código de bloque que tiene una longitud de palabra fija n, un código convolucional no tiene un tamaño de bloque particular. Sin embargo, los códigos convolucionales son en muchas ocasiones forzados a una estructura de bloque por un *truncamiento periódico*. Esto requiere un número de bits ceros para ser anexados en el final de la secuencia de datos de entrada, con el propósito de limpiar los registros de desplazamiento del codificador de los bits de datos. Debido a que los ceros anexados no acarrean información, la razón efectiva de código cae debajo de s/n. Para mantener la razón de código tan cercano a k/n el periodo de truncamiento es generalmente hecho tan largo como el caso lo permita.

Una manera para representar el codificador es especificar una serie de vectores de conexión, uno para cada uno de los n sumadores módulo-2. Cada vector tiene dimensión **K** y describe la conexión del registro de desplazamiento del codificador al sumador módulo-2. Un uno en la i-ésima posición del vector indica que la correspondiente etapa en el registro es conectada a el sumador módulo-2,y un cero en una posición dada indica que no existe conexión entre la etapa y el sumador módulo-2. Para el codificador del ejemplo en la figura 6.3, podemos escribir el vector de conexión $\mathbf{g}_1$ para las conexiones de arriba y **g2** para las conexiones de abajo como sigue:

$$\mathbf{g}_1 = 1\ 1\ 1$$
$$\mathbf{g2} = 1\ 0\ 1$$

Considere que un vector de mensaje **m** = 1 0 1 es codificado convolucionalmente con el codificador mostrado en la figura 1.3. El mensaje de tres bits es introducido, un bit a al vez, en los tiempos t_1, t_2, y t_3, como lo muestra la figura 1.4. Posteriormente, (K – 1) = 2 ceros son introducidos en los tiempos t_4 y t_5 para limpiar el registro y entonces asegurar que el final del mensaje es desplazado totalmente fuera del registro. La secuencia de salida es 1 1 1 0 0 0 1 0 1 1 , donde el símbolo mas a la izquierda representa el principio de la transmisión. La secuencia de salida entera, incluyendo el símbolo codificado que resulto de limpiar el registro, es necesario para decodificar el mensaje.

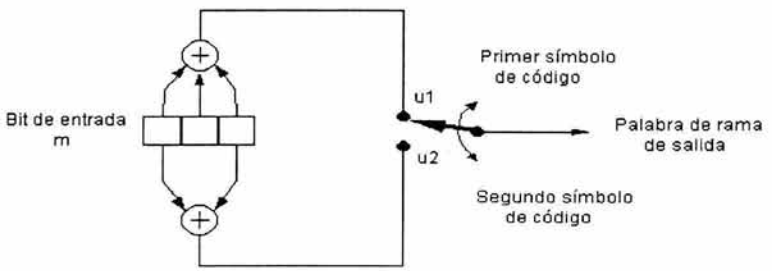


Figura 1.3 Codificador convolucional (razón ½ , K= 3)

Para sacar totalmente el mensaje fuera del codificador se requiere una cantidad de ceros igual a K -1 bits, donde K es el número de etapas del registro. Otra entrada cero es mostrada en un tiempo t_6, se puede verificar que la correspondiente palabra de rama de salida es 00.

1.2.3 El Diagrama de árbol

Aunque el diagrama de estado caracteriza completamente el codificador, uno no puede fácilmente usarlo para un seguimiento de las transiciones del codificador como una función del tiempo debido a que el diagrama no puede representar la historia a través del tiempo. El diagrama de árbol adiciona esta dimensión temporal al diagrama de estados. El diagrama de árbol para el codificador convolucional mostrado en la figura 1.3 es ilustrado en la figura 1.6. En cada sucesivo tiempo de bit de entrada el procedimiento de codificación puede ser descrito por medio de ir atravesando en diagrama de izquierda a derecha, cada rama del árbol describe una palabra de rama de salida. La regla para ir moviéndose a través de las ramas es como sigue: Si el bit de entrado es un cero, su rama asociada es encontrada moviéndose a la rama hacia la derecha más próxima, hacia arriba. Si el bit de entrada es un

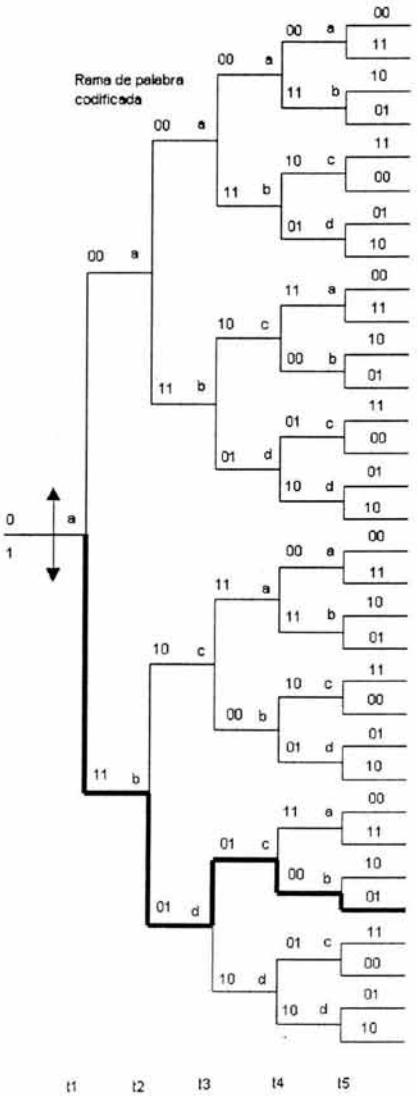


Figura 1.4 Representación del codificador (razón ½ , K= 3)

uno, su rama es encontrada moviéndose hacia la rama más próxima hacia la derecha moviéndose hacia abajo.

Asumiendo que el contenido inicial del codificador son todos ceros, el diagrama muestra que si el primer bit de entrada es un cero, la palabra de rama de salida es 00 y, si el primer bit de entrada es un uno, la palabra de rama de salida es 11. Similarmente, si el primer bit de entrada es un una y el segundo bit de entra es un cero, la segunda palabra de rama de salida es 10 ó si el primer bit de entrada es un uno y el segundo bit de entrada en

un uno, la segunda palabra de rama de salida 01. Siguiendo este procedimiento podemos ver que la secuencia de entrada 1 1 0 1 1 traza una línea gruesa dibujada en el diagrama de árbol de la figura 1.4. Este camino corresponde a la siguiente secuencia de salida: 1 1 0 1 0 1 0 0 0 1.

La dimensión de tiempo sumada en este diagrama de árbol (comparada con el diagrama de estados) permite a uno hacer una descripción del codificador como una función de una secuencia particular de entrada. Aunque, ¿Se puede observar un problema al tratar de usar un diagrama de árbol para describir una secuencia de cualquier longitud? El número de ramas se incrementa como una función de 2^l, donde l es el numero de bits en la secuencia de entrada. El diagrama se saldría rápidamente del papel.

1.2.4 El Diagrama de trellis o de enrejado

De observar el diagrama de árbol de la figura 1.4 mostrado para el ejemplo, la estructura se repite a partir del tiempo t_4, después de la tercera rama (en general, la estructura del árbol se repite después de kK ramas, donde K es la longitud restringida). Se etiqueta cada nodo en el árbol de la figura 1.4 de tal manera que correspondan con los cuatro estados posibles en el registro de desplazamiento, como sigue: a = 00, b = 10. c = 01 y d =11. La primer rama de la estructura de árbol, en el tiempo t_1, produce un par de nodos etiquetados como a y b. En cada rama sucesiva el número de nodos es el doble. La segunda rama, en el tiempo t_2, resulta en cuatro nodos etiquetados como a, b, c y d. Después de la tercera rama hay un total de ocho nodos; dos de los cuales son etiquetados con a, dos son etiquetados con b, dos con c y dos con d. Se puede observar que todas las ramas que emanan de dos nodos del mismo estado generan palabras de ramas con secuencias idénticas.

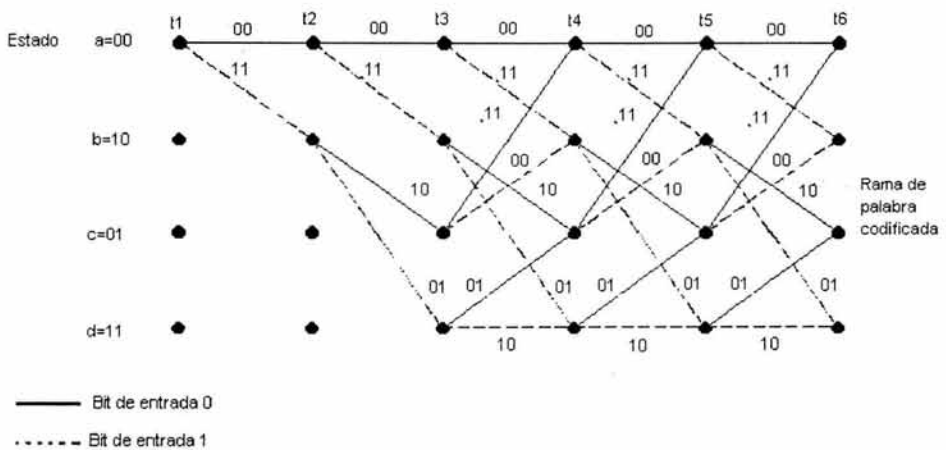


Figura 1.5 Diagrama de Trellis del codificador (razón ½ , K= 3)

En este sentido, las mitades superior e inferior del diagrama del árbol son idénticas. La razón para lo anterior es obvia de examinar al codificador de la figura 1.3. Como los

cuatro bits de estrada son introducidos al codificador en la izquierda, el primer bit de entrada sale en la derecha y no tiene influencia en las palabras de rama de salida. Por esta razón, la secuencia de entrada 1 0 0 x y ... y 0 0 0 x y . . .. donde el bit más a la izquierda es el primer bit que entra, generan la misma palabra de rama después de construir la (K= 3) etapa del diagrama. Esto significa que dos nodos cualesquiera que tienen la misma etiqueta de estado, en el mismo tiempo t_i, pueden fusionarse debido a que sus sucesivas ramas con indistinguibles. Si hacemos esto a la estructura de árbol de la figura 1.4 obtenemos otro diagrama, llamado el diagrama de enrejado o diagrama de trellis, explotando su respectiva estructura, provee una más manejable descripción del codificador mas que lo hace el diagrama de árbol. El diagrama de trellis para el codificador convolucional de la figura 1.3 es mostrado en la figura 1.5

En el dibujo del diagrama de trellis, usamos la misma convención que usamos en el diagrama de estados (que una línea sólida denota la salida generada por un bit de entrada cero, y una línea punteada denota la salida generada por un bit de entrada uno). Los nodos del trellis caracterizan los estados del codificador. El enrejado en nuestro ejemplo asume una estructura fija periódica después alcanzar una longitud de enrejado 3 (en el tiempo t_4). En el caso general, la estructura fija prevalece después de alcanzar una longitud de K. Después de este punto, cada uno de los estados puede ser introducido por cualquiera de los dos estados que le preceden. También, cada uno de los estados puede cambiar a uno o dos estados. De las dos ramas salientes, una corresponde a un bit de entrada cero y la otra corresponde a un bit de entra uno. En la figura 1.5 las palabras de rama de salida correspondientes a una transición de estado parecen como etiqueta sobre las ramas del enrejado.

1.3 Formulación del problema de la decodificación convolucional.

1.3.1 Decodificación de máxima probabilidad

Si todas las secuencias del mensaje de entrada son igualmente probables, un decodificador que tiene la mínima probabilidad de error es uno que compara las probabilidades condicionales, también llamadas funciones de similitud, $\mathbf{P(Z|U^{(m)})}$ [1, Pag 327], donde $\mathbf{Z}$ es la secuencia recibida y $\mathbf{U}^{(m)}$ es una de las posibles secuencias transmitidas, y escoge la máxima. El decodificador escoge $\mathbf{U}^{(m)}$ si

$$\mathbf{P(Z|U^{(m)})} = \max \mathbf{P(Z|U^{(m)})} \text{ para toda } \mathbf{U}^{(m)} \qquad (1.1)$$

El concepto de máxima probabilidad o similitud, como esta expresado en la ecuación (1.1), es un fundamental desarrollo de la teoría de decisiones; es la formalización del "sentido común" que nos lleva a tomar decisiones cuando hay un conocimiento estadístico de las posibilidades. En el tratamiento de la demodulación binaria hay solo dos señales posibles igualmente probables, s1(t) o s2(t), que debieron ser transmitidas. Por eso, para hacer la decisión binario con máxima probabilidad, dada una señal recibida, esto quiere decir que para decidir que s1(t) fue transmitida si

$$p(z|s1) > p(z|s2)$$

de otro modo, decide que s2(t) fue transmitida. El parámetro z representa z(T), la salida del receptor en un tiempo de duración de símbolo t=T. Aunque, cuando aplicamos la máxima probabilidad al problema de la decodificación convolucional, hay típicamente una multitud de posibles secuencias de palabras codificadas que pudieron ser transmitidas. Para ser más específicos, una secuencia de una palabra codificada de L-bits es un miembro de un conjunto de 2^L posibles secuencias. Entonces, en el contexto de la máxima probabilidad, podemos decir que el decodificador escoge una particular $\mathbf{U}^{(m)}$ como la secuencia transmitida si la probabilidad $P(Z|\mathbf{U}^{(m)})$ es más grande que las probabilidades de todas las otras posibles secuencias transmitidas. Como un decodificador óptimo, el cual minimiza la probabilidad de error (para el caso donde todas las secuencias transmitidas son igualmente probables), es conocido como un decodificador de máxima probabilidad. Las funciones de probabilidad son dadas o calculadas de las especificaciones del canal.

Podemos asumir que el ruido es ruido blanco Gaussiano aditivo con media cero y el canal es sin memoria, lo cual significa que el ruido afecta a cada símbolo del código en forma independiente a los otros símbolos. Para un codificador convolucional de razón 1/n, podemos entonces expresar la probabilidad, $P(Z|\mathbf{U}^{(m)})$ [1, pag. 328]como sigue:

$$\mathbf{P(Z|U^{(m)}) = \prod_{i=1}^{\infty} P(Z_i|U_i^{(m)}) = \prod_{i=1}^{\infty}\prod_{j=1}^{n} P(z_{ji}|u_{ji}^{(m)})} \qquad (1.2)$$

Donde Z_i es la i-ésima rama de la secuencia recibida Z, , $\mathbf{U}_i^{(m)}$ la i-ésima rama de una particular secuencia de palabra codificada $\mathbf{U}_i^{(m)}$, z_{ji} el j-ésimo símbolo de código de Zi, y $u_{ji}^{(m)}$ el j-ésimo símbolo de código de $\mathbf{U}_i^{(m)}$, cada rama comprende *n* símbolos de código. El problema de decodificar consiste en escoger un camino a través del trellis de la figura 1.7 (cada posible camino define una palabra codificada) de tal manera que

$$\mathbf{\prod_{i=1}^{\infty}\prod_{j=1}^{n} P(z_{ji}|u_{ji}^{(m)})} \text{ sea máximo} \qquad (1.3)$$

Generalmente, es más conveniente, desde el punto de vista del cómputo, usar el logaritmo de la función de probabilidad debido a que esta permite la suma, en lugar de la multiplicación, de los términos. Podemos usar esta transformación porque el logaritmo es una función monótonamente creciente y esto no altera el resultado final en nuestra selección de la palabra codificada. Podemos definir el logaritmo de la función de probabilidad $\gamma_u(m)$ como

$$\gamma_u(m) = \log P(\mathbf{Z}|\mathbf{U}^{(m)}) = \sum_{i=1}^{\infty} \log P(Z_i|U_i^{(m)}) = \sum_{i=1}^{\infty}\sum_{j=1}^{\infty} \log P(z_{ji}|u_{ji}^{(m)}) \qquad (1.4)$$

El problema de decodificar ahora consiste de escoger un camino a través del árbol de la figura 1.6 o el trellis de la figura 1.7 de tal manera que $\gamma_u(m)$ sea maximizada. Para la decodificación de los códigos convolucionales, cualquiera, el diagrama de árbol o el de

estructura de enrejado puede ser usado. En el diagrama de árbol, debido a que el número de posibles secuencias para una secuencia de longitud L es 2^L , la decodificación de máxima probabilidad de una secuencia recibida de longitud L, usando el diagrama de árbol, requiere de "fuerza bruta" o una comparación exhaustiva de 2^L posibilidades, que representan todas las posibles diferentes palabras codificadas que pudieron ser transmitidas. Por esto no es práctico considerar la decodificación de máxima probabilidad con la estructura de árbol. Se mostrará más adelante que con el uso de la representación de enrejado del código, es posible configurar un decodificador el cual puede descartar los caminos que no tienen posibilidad de ser candidatos para la secuencia de máxima probabilidad. Se escoge el camino decodificado dentro un conjunto reducido de *caminos sobrevivientes.* Un decodificador puede ser óptimo en el sentido de que el camino decodificado es el mismo que el camino decodificado obtenido por medio de un decodificador de máxima probabilidad que usa "fuerza bruta, pero el pronto rechazo de caminos con poco probabilidad reduce la complejidad del decodificador.

Hay muchos algoritmos que dan una solución apropiada al problema de la decodificación de máxima probabilidad, cada uno de esos algoritmos es implementado en ciertas aplicaciones especiales, pero todos son sub-óptimos. En contraste, el *algoritmo de decodificación de Viterbi* desarrolla decodificación con máxima probabilidad y es por lo tanto óptimo[1, pag.329]. Esto no implica que el algoritmo de Viterbi es el mejor para toda aplicación; hay muchas restricciones impuestas por la complejidad del hardware.

1.3.2 Modelos del Canal: Decisión dura contra decisión suave

Anteriormente se especificó un algoritmo que determina la decisión con máxima probabilidad, ahora describiremos el canal. La secuencia de la palabra codificada $\mathbf{U}^{(m)}$, construida con palabras de rama, cada una construida con *n* símbolos de código, puede ser considerada como una cadena sin fin, en oposición a un código de bloque, en el cual la fuente de datos y sus palabras codificadas son partidas en bloques de tamaño preciso. La palabra codificada en la figura 1.1 emana del codificador convolucional y entra en el modulador, donde los símbolos de código son transformados en formas de onda. La modulación puede ser en banda base (en forma de pulsos) o en pasa banda (PSK o FSK). En general, l símbolos en un tiempo, donde l es un entero, son mapeados en formas de onda $s_i(t)$, donde $i = 1, 2, \ldots, M = 2^l$, cuando l = 1, el modulador mapea coda símbolo de código en una forma de onda binaria. El canal sobre el cual la forma de onda es transmitida se asume que corrompe a la señal con ruido Gaussiano. Cuando la señal es recibida, es procesada primeramente por el demodulador y luego por el decodificador.

Considere que una señal binaria, transmitida sobre un intervalo de símbolo (0, T), es representado por $s_1(t)$ para un uno binario y $s_2(t)$ para un cero binario. La señal recibida es $r(t) = s_1(t) + n(t)$, donde n(t) es un ruido Gaussiano con media cero. Se puede hacer la detección de r(t) en términos de dos pasos básicos. En el primer paso, la forma de onda recibida es reducida a un simple número, $z(T) = a_i + n_o$, donde a_i es la componente de señal de z(T) y n_o es la componente de ruido. La componente de ruido, n_o, es *variable aleatoria Gaussiana* con media cero, y entonces z(T) es una variable aleatoria Gaussiana con media a1 o a2 dependiendo de si un cero o un uno binario fue enviado. En un segundo paso del proceso de detección una decisión es hecha para saber cual señal fue transmitida, con base a una comparación de z(T) con un umbral de decisión. Las probabilidades de z(T), $p(z| s_1(t))$

y p(zl $s_2(t)$) son mostradas en la figura 1.6, y son etiquetadas como probabilidad de s_1 y probabilidad de s_2. El demodulador de la figura 1.6, convierte el conjunto de variables aleatorias {z(T)}, en una secuencia de código **Z** y pasa esta secuencia al decodificador. La salida del demodulador puede ser configurada en una variedad de formas. Esto puede ser implementado para hacer una decisión firme o dura de tal manera que z(T) representa un cero o un uno. En este caso, la salida del demodulador es cuantizada en dos niveles, cero y uno, y alimenta al decodificador. Debido a que el decodificador opera con base a decisiones duras hechas por el demodulador, la decodificación es llamada *decodificación de decisión dura.*

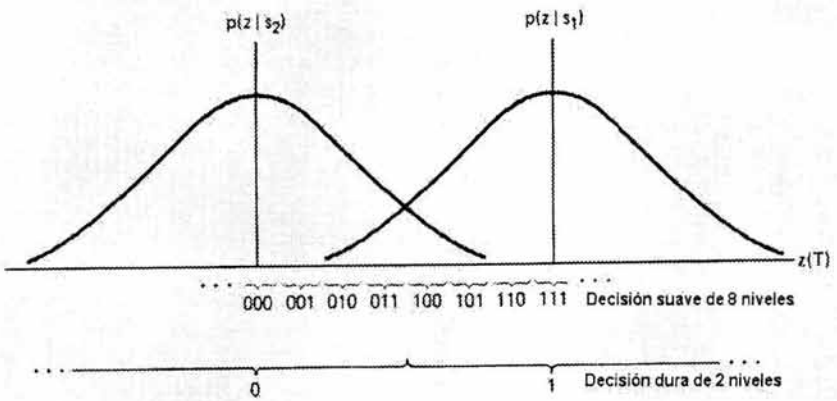


Figura 1.6 Decisiones dura y suave para la decodificación

El demodulador puede ser también configurado para alimentare al *decodificador con un valor cuantizado de z(T) de más de dos nivele,* o con un valor no cuantizado o análogo de z(T). Tal implementación alimenta al decodificador con más información que la que se provee en el caso de decisión dura. Cuando el nivel de cuantización de la salida del demodulador es más grande que dos, la decodificación es llamada decodificación con decisión suave. Ocho niveles (3 bits) de cuantización son ilustrados en el eje de las abscisas de la figura 1.6. Cuando el demodulador envía una decisión binaria dura al decodificador, este envía un solo símbolo binario. Cuando el demodulador envía una decisión binaria suave, cuantización a ocho niveles, este envía al decodificador una palabra de 3 bits describiendo un intervalo en z(T). En efecto, enviando una palabra de tres bits en lugar de un solo símbolo binario es equivalente a enviar al decodificador una medida de confianza sobre el símbolo de código. Haciendo referencia a la figura 1.6, si el demodulador envía 1 1 1 al decodificador, esto es equivalente a decir que el símbolo de código es uno con una muy alta confianza, mientras que enviando un 1 0 0 es equivalente a decir que el símbolo de código es un uno con muy baja confianza. Debe quedar claro que a fin de cuentas, todas las decisiones acerca del mensaje fuera del decodificador deben ser una decisión dura; de otro modo, se podría ver a la computadora desplegando el mensaje: " creo que esto es un 1", "creo que esto es un cero" y mensajes de este tipo. La idea detrás

del demodulador que no hace una decisión dura y envía mas datos (decisión suave) al decodificador puede pensarse como un paso intermedio que provee al decodificador con más información, que el decodificador debe de usar para reconstruir la secuencia del mensaje (con una mejor desempeño de error que este puede presentar en el caso de la decodificación con decisión dura).

Para un canal Gaussiano, una cuantización con ocho niveles resulta en una mejora en el desempeño de aproximadamente 2 dB en la razón señal a ruido requerida, comparada con la cuantización de dos niveles. Esto significa que la decodificación de decisión suave de ocho niveles puede proveer la misma probabilidad de error en un bit que provee la decodificación de decisión dura, pero que requiere 2 dB menos en la relación señal a ruido para el mismo desempeño. La forma analógica (o cuantización de niveles infinitos) resulta en 2.2 dB de mejora en el desempeño sobre la cuantización de dos niveles: Por tal motivo, la cuantización de ocho niveles resulta en una pérdida de aproximadamente 0.2 dB comparada a cuantización de niveles infinitos[1,pag 331]. Por esta razón, la cuantización de mas de ocho niveles puede dar muy poca mejora en el desempeño. ¿Que precio hay que pagar por el mejoramiento en el desempeño del decodificador decisión suave? En el caso de la decodificación de decisión dura, un solo bit es usado para describir cada símbolo de código, mientras que para la cuantización de ocho niveles la decodificación de decisión suave son usados 3 bits para describir cada símbolo de código: Por eso, tres veces la cantidad de datos deben ser manejados durante el proceso de decodificación. Aunque, el precio que se paga por la decodificación de decisión suave, es un incremento en el tamaño de la memoria requerida en el decodificador (y probablemente un costo en la velocidad).

Los algoritmos de decodificación de bloque y los algoritmos de decodificación convolucional han sido observados operando con decisiones duras y suaves. Sin embargo, la decodificación de decisión suave no es generalmente usada con códigos de bloque porque estos son considerablemente más difíciles de implementar que aquellos con decodificación de decisión dura. El mayor uso de la decodificación con decisión suave es con algoritmo de decodificación convolucional de Viterbi, debido a que con la decodificación de Viterbi, las decisiones suaves representan solo un trivial incremento en el procesamiento de cómputo.

1.3.2.1 Canal binario simétrico

Un canal binario simétrico (BSC) es canal discreto sin memoria que tiene entrada binaria con salida alfabética y probabilidad de transición simétrica. Esto puede ser descrito por las siguientes probabilidades condicionales

$$\begin{aligned} P(0|1) &= P(1|0) = p \\ P(1|1) &= P(0|0) = p - 1 \end{aligned} \tag{1.5}$$

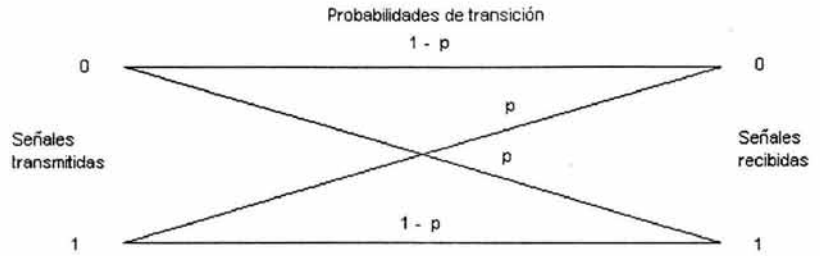


Figura 1.7 Canal simétrico binario (Canal de decisión dura).

Como es ilustrado en la figura 1.7. La probabilidad de que un símbolo de salida sea diferente del símbolo de entrada es *p*, y la probabilidad de que el símbolo de salida sea idéntico al símbolo de entrada es (1 – p) . El BSC es un ejemplo de un canal de decisión dura, lo cual significa que, aun con valores de señal continuos que pueden ser recibidos por el demodulador, un BSC permite solo decisiones firmes de tal manera que cada símbolo de salida del demodulador, *zji*, como se muestra en la figura 1.1, consiste de uno de los dos valores binarios. Los índices de *zji* pertenecen al j-ésimo símbolo de la i-ésima palabra de rama, Z_i . Entonces el demodulador alimenta la secuencia Z = { Z_i } en el decodificador.

Sea $\mathbf{U}^{(m)}$ una palabra codificada transmitida sobre un BSC con probabilidad de error de símbolo *p,* y sea **Z** su secuencia decodificada recibida correspondiente. Como fue notado previamente, un decodificador de máxima probabilidad escoge la palabra codificada $\mathbf{U}^{(m)}$ que maximiza la probabilidad P(Z|$\mathbf{U}^{(m)}$) o su logaritmo. Para un BSC, esto es equivalente a escoger la palabra codificad $\mathbf{U}^{(m)}$, que tiene la *distancia de Hamming* más pequeña a **Z**. Entonces la distancia de Hamming es una apropiada medida para describir la cercanía o que tan parecidas son $\mathbf{U}^{(m)}$ y **Z**. De todas las secuencias posibles transmitidas, $\mathbf{U}^{(m)}$, el decodificador escoge la secuencia $\mathbf{U}^{(m)}$ para la cual la distancia con **Z** es mínima.

Suponga que $\mathbf{U}^{(m)}$ y **Z** son cada una secuencia de longitud de L bits y que ellas difieren en d_m posiciones (esto es, la distancia de Hamming entre $\mathbf{U}^{(m)}$ y **Z** es d_m). Entonces, debido a que el canal es asumido con sin memoria, la probabilidad de que esta $\mathbf{U}^{(m)}$ fue transformada a la específica **Z** recibida, y a una distancia d_m de esta, puede escribirse como

$$P(Z|\,\mathbf{U}^{(m)}) = p^{\,dm}(\,1-p)^{\,L-dm} \tag{1.6}$$

y el logaritmo de la función de probabilidad es

$$\text{Log } P(Z|\,\mathbf{U}^{(m)}) = -d_m \log((\,1-p)/p) + L \log(1-p) \tag{1.7}$$

Si calculamos esta cantidad para cada posible secuencia transmitida, el segundo término debe ser constante en cada caso. Asumiendo que p < 0.5, podemos expresar la ecuación anterior como

$$\text{Log } P(Z|\,\mathbf{U}^{(m)}) = -\,A\,d_m - B \tag{1.8}$$

Donde A y B son constante positivas. Entonces, escogiendo la palabra codificada $\mathbf{U}^{(m)}$ tal que la distancia de Hamming., d_m, a la secuencia recibida Z es mínima, esto corresponde a maximizar la probabilidad o el logaritmo de la probabilidad. Consecuentemente, sobre un BSC, el logaritmo de la probabilidad es convenientemente reemplazado con la distancia de Hamming, y un decodificador de máxima probabilidad debe escoger, en el diagrama de árbol o en el enrejado, el camino al cual corresponda a la secuencia $\mathbf{U}^{(m)}$, que tiene la distancia de Hamming mínima a la secuencia recibida **Z**.

1.3.2.2. Canal Gaussiano

Para un canal Gaussiano, cada símbolo de salida del demodulador, z_{ji} no puede ser etiquetado como una detección con decisión correcta o incorrecta. Enviando el decodificador decisiones suaves que pueden ser vistas como el envío de una familia de probabilidades condicionales de los diferentes símbolos. Puede mostrarse que la maximización de $P(Z| \mathbf{U}^{(m)})$ es equivalente a maximizar el producto interno entre la secuencia dela palabra codificada $\mathbf{U}^{(m)}$ (que consiste de símbolos binarios), y la secuencia recibida de valores análogos, Z. Entonces el decodificador escoge la palabra codificada $\mathbf{U}^{(m)}$ si esta maximiza a:

$$\sum_{i=1}^{\infty} \sum_{j=1}^{n} z_{ji}\, u_{ji}^{(m)} \tag{1.9}$$

Esto es equivalente a escoger la palabra codificada $\mathbf{U}^{(m)}$ que es la más cerrada en la *distancia Euclidiana* a **Z**[1, pag. 332]. Aunque los canales con decisión suaves y duras requieren de diferentes mediciones, el concepto de escoger la palabra codificada $\mathbf{U}^{(m)}$ que es la secuencia recibida más cercana, Z, es la misma en ambos casos. Para implementar la maximización exacta de la ecuación 1.9, el decodificador debe tener la capacidad de manejar operaciones aritméticas con valores análogos. Esto resulta impractico porque el decodificador es generalmente implementado digitalmente. Entonces es necesario cuantizar los símbolos recibidos z_{ji}. La ecuación 1.9 es la versión discreta de la correlación entre la forma de onda recibida r(t) con una forma de onda de referencia, $s_i(t)$. El canal cuantizado Gaussiano, típicamente llamado como un canal de decisión suave, es el modelo de canal asumido para el decodificador de decisión suave descrito anteriormente.

1.3.3 El Algoritmo de Decodificación Convolucional de Viterbi

Por simplicidad, tomamos un BSC; entonces la distancia de Hamming es la medida de distancia más apropiada. El decodificador para este ejemplo esta mostrado en la figura 1.3, y el diagrama de trellis del decodificador es mostrado en la figura 1.5. La idea básica detrás del proceso de la decodificación puede ser mejor entendida examinando el diagrama de la figura 1.5 conjuntamente con el diagrama de la figura 1.8. Para el decodificador trellis es conveniente etiqueta cada rama en el tiempo t_i con la distancia de Hamming entre los símbolos de código recibidos y la correspondiente palabra de rama del codificador trellis. El ejemplo en la figura 1.8, muestra una secuencia de mensaje, **m**, la correspondiente secuencia de palabra codificada, **U**, y una secuencia recibida corrompida por ruido, **Z** = 11 01 01 10 01 . . . . Las palabras de rama que se observan en el ramas del *codificador trellis*

caracterizan al codificador de la figura 1.3, y son conocidas antes del codificador y el decodificador. Estas palabras de rama del codificador son los símbolos de código que podemos esperar que vengan de la salida del codificador como resultado de cada una de las transiciones de estado. Las etiquetas en las ramas del decodificador son acumuladas por el decodificador *en el vuelo.* Esto es, como los símbolos de código son recibidos , cada rama del decodificador de trellis es etiquetado con una medida de similitud (distancia de Hamming) entre los símbolos de código recibidos y cada una de las palabras de rama de aquel intervalo de tiempo. De la secuencia recibida, **Z**, en la figura 1.8, podemos ver que los símbolos de código recibidos en el tiempo t_1 son 1 1. Para etiquetar las ramas en el tiempo t1 con la distancia de Hamming, vemos en el codificador de la figura 1.5. Aquí vemos que una transición de estado 00 →00 da palabra de rama de salida 00, pero nosotros recibimos 11.

Secuencia de datos de entrada	m	1	1	0	1	1	...
Palabra codificada transmitida	U	11	01	01	00	01	...
Secuencia recibida	Z	11	01	01	10	01	...

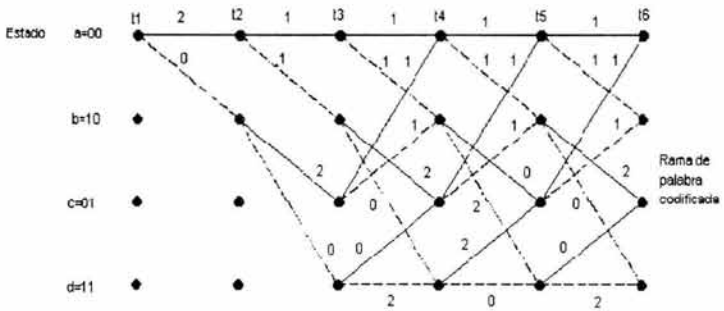


Figura 1.8 Diagrama de Trellis del codificador (razón ½ , K= 3)

Por eso, en el decodificador de trellis se etiqueto a la transición de estado 00 →00 con la distancia de Hammig entre ellos, esto es, observando en el codificador de trellis nuevamente, vemos que la transición de estado 00 →10 de una palabra de rama de salida de 11, el cual corresponde exactamente con el símbolo de código que recibimos en el tiempo t_1. Pos esto, en el decodificador de trellis, etiquetamos a la transición de estado 00 →10 con la distancia de Hamming de 0. Podemos continuar etiquetando a las ramas del decodificador de trellis de esta forma con los símbolos recibidos en cada tiempo t_i. El algoritmo de decodificación usa esta medida de la distancia de Hamming para encontrar el camino *más probable* (distancia mínima) a través de las trellis.

La base de la *decodificación de Viterbi* es la siguiente observación: Si dos caminos cualquiera en el enrejado llegan a un mismo estado, uno de ellas puede ser siempre eliminado en la búsqueda del camino más óptimo. Por ejemplo, la figura 1.9 muestra dos caminos llegando en un tiempo t_5 al estado 00. Definamos la medida de camino acumulado

de Hamming de un camino dado en un tiempo ti como la suma de las distancias de Haming de rama para ese camino construido hasta el tiempo ti. En la figura 1.9 el camino de arriba tiene una medida de 4; y el de abajo tiene una medida de 1. El camino de arriba no puede ser una porción del camino óptimo porque el camino de abajo, el cual entra al mismo

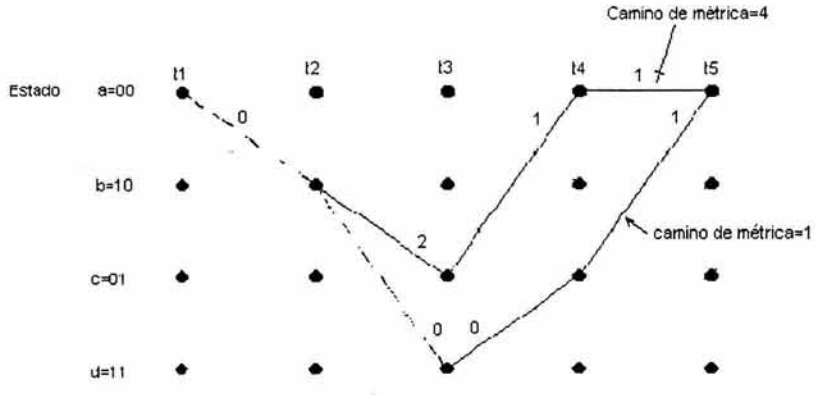
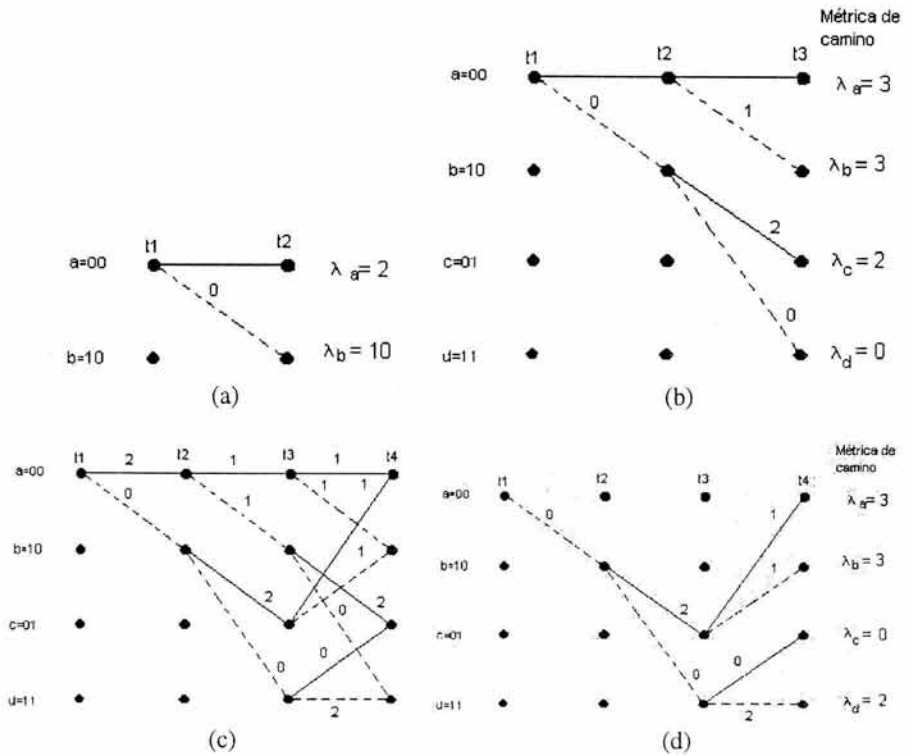


Figura 1.9 Métricas de camino para dos caminos que convergen

estado, tiene una métrica más baja. El presente estado resume la historia del codificador en el sentido de que los estados previos no pueden afectar futuros estados o futuras salidas de rama.

En cada tiempo t_i, hay 2^{k-1} estados en el enrejado, donde K es la longitud restringida, y cada estado puede ser *alcanzado por dos caminos*. La decodificación de Viterbi, consiste de computar las medidas de los dos caminos que entran en cada estado y eliminar uno de ellos. Este computo es hecho para cada uno de los 2^{k-1} nodos en un tiempo t_i; entonces el decodificador llega al tiempo t_{i+1} y repite el proceso. Los primeros pocos pasos de nuestro ejemplo de decodificación son como sigue (ver la figura 1.10). Asumimos que la secuencia de datos de entrada **m**, la palabra codificada **U** y la secuencia recibida **Z** son las mostradas en la figura 1.8. Asumimos que el decodificador conoce el estado inicial correcto del enrejado. (Esta suposición no es necesaria en la práctica, pero simplifica la explicación) En el tiempo t_1 el símbolo de código recibido es 11. Desde el estado 00 las únicas posibilidades de transiciones son al estado 00 o al estado 10. Como se muestra en la figura 1.10a. La transición de estado 00 $\rightarrow$00 tiene una métrica de rama de 2; la transición de estado 00 $\rightarrow$10 tiene una métrica de rama de 0. En el tiempo t_2 hay dos posibles ramas dejando cada estado, como se muestra en la figura 1.10b. la medida acumulada de camino de esas ramas, son etiquetadas con λ_a, λ_b, λ_c y λ_d, correspondientes al estado terminal. En el tiempo t3 en la figura 1.10c hay nuevamente dos ramas saliendo de cada estado. Como resultado, hay dos caminos entrando en cada estado en el tiempo t4. Como se hizo notar previamente, un camino entrando en cada estado puede ser eliminado, aquel que tenga la más grande medida de camino acumulado. Aquellas medidas que sean iguales de dos caminos que entran, unos de los dos puede ser eliminado usando alguna regla arbitraria. El camino sobreviviente en cada estado es mostrado en la figura 1.10d. En este punto del proceso de la decodificación, hay solo un comino sobreviviente entre los tiempos t1 y t2.

Entonces, el decodificador puede ahora decidir que la transición de estado que ocurrió entre los tiempos t1 y t2 fue 00 →10. Debido a que esta transición es producida por una entrada de bit uno, el decodificador tiene una salida de 1 en el primer bit decodificado. Aquí podemos ver que la decodificación de la rama sobreviviente es facilitada teniendo un dibujo de las ramas con líneas sólidas para entrada de ceros y una línea punteada para entrada de unos. Notar que el primer bit no fue decodificado hasta computar la medida de camino después de haber llegado a una mayor profundidad en el enrejado. Para una típica implementación del decodificador, esto representa un retraso de decodificación en cual puede ser tanto como cinco veces la longitud restringida en bits.



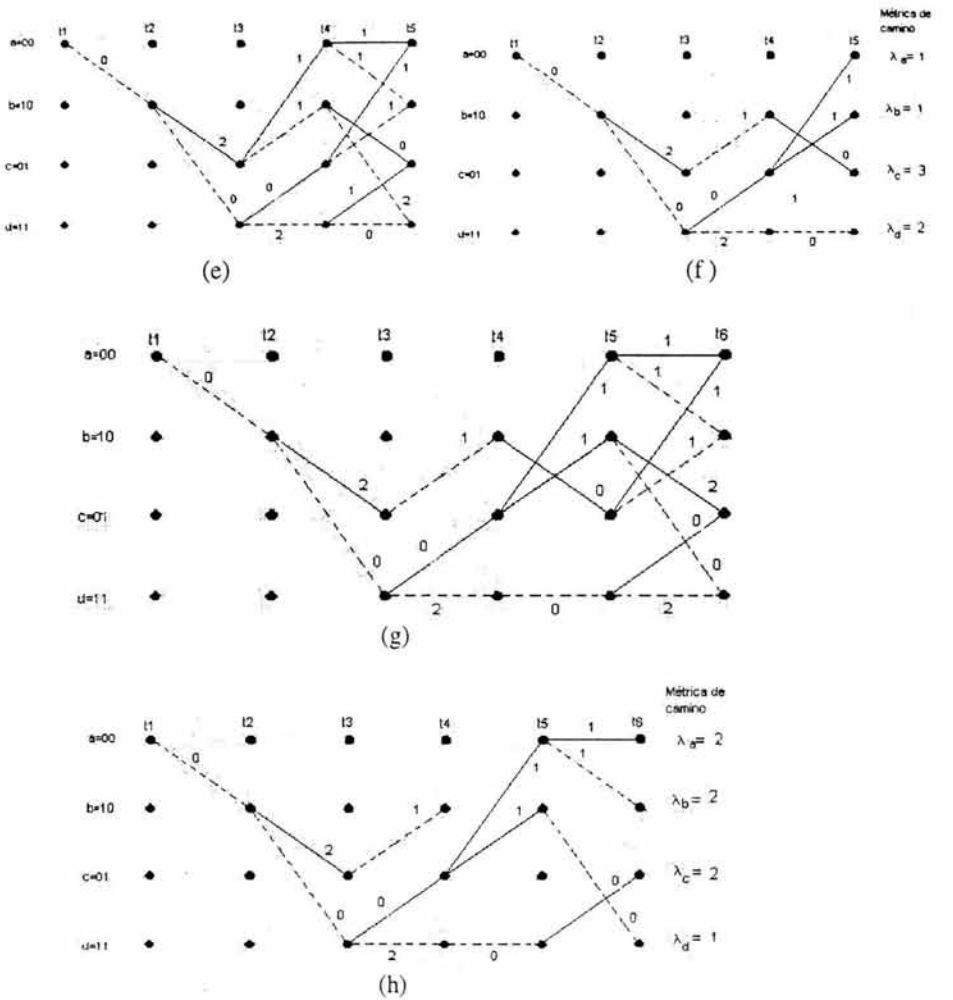


Figura 1.10 Selección de los caminos sobrevivientes. (a) Sobrevivientes en t2. (b) Sobrevivientes en t3. (c)Comparaciones de métricas en t4. (d)Sobrevivientes en t4. (e) Comparaciones de métricas en t5. (f)Sobrevivientes en t5. (g) Comparaciones de métricas en t6. (h) Sobrevivientes en t6.

En cada paso exitoso en el proceso de decodificación, debe haber siempre dos posibles caminos que entran en cada estado; uno de los dos debe ser eliminado por comparación de la métrica de camino. La figura 1.10e muestra el siguiente paso en el proceso de decodificación. Nuevamente, en el tiempo t_5 hay dos caminos entrando en ada estado, y uno de cada par debe ser eliminado. La figura 1.10f muestra los caminos sobrevivientes en el tiempo t_5. Notar que en muestro ejemplo no podemos hacer todavía una decisión sobre en segundo bit de entrada porque hay todavía dos caminos dejando el nodo del estado 10 en el tiempo t_2. En el tiempo t_6 en la figura 1.10g podemos ver

nuevamente el patrón de caminos, y un al figura 1.10h podemos ver los caminos sobrevivientes en el tiempo t_6. También, en la figura 1.10h la salida del decodificador es uno, y es el segundo bit decodificado, correspondiente al único camino sobreviviente entre t_2 y t_3. El decodificador continúa de esa forma para avanzar más profundo dentro del diagrama de trellis y hacer decisiones sobre los bits de entrada mediante la eliminación de todos los caminos excepto uno.

1.3.4 Memoria de caminos y Sincronización

Los requerimientos de memoria del decodificador de Viterbi crecen en forma exponencial en función de la longitud restringida K. Para un código con razón 1/n el decodificador almacena un conjunto de 2k-1 caminos después de cada paso de decodificación. Con alta probabilidad, estos caminos no deben ser mutuamente excluyentes muy atrás de la presente profundidad de decodificación. Todos los 2k-1 caminos tienden a tener un origen en común, el cual se dispersa a varios estados. Entonces, si el decodificador almacena suficiente historia de los 2k-1 caminos, el origen de los bits en todos los caminos debe ser el mismo. Una implementación simple de un decodificador, contiene entonces una cantidad fija de historia de caminos y manda como salida el bit que se procesó primero de un camino arbitrario, cada vez que este salta un nivel más profundo en el diagrama de trellis. La cantidad de almacenamiento de caminos requerida, u, es

$$u = h2^{k-1} \qquad (1.10)$$

Donde h es la longitud de la historia de camino del bit de información por estado. Una recomendación, que minimiza el valor de h[1, pag 337], usa el bit más antiguo en el camino más probable como salida del decodificador, en lugar del bit más antiguo de un camino arbitrario. Esta demostrado que el valor de h de 4 o 5 veces la longitud restringida del código es suficiente para un desempeño óptimo del código. Los requerimientos de almacenaje, u, es la limitación básica en la implementación de los decodificadores del Viterbi, Actualmente se encuentra un límite en la implementación de los decodificadores, a una longitud restringida de alrededor de K=10. Esfuerzos por incrementar la ganancia de código mediante más incremento de la longitud restringida, es acompañada por el incremento exponencial de los requerimientos de memoria (y complejidad) que siguen de la ecuación 1.10.

Capítulo 2

Simulación de la Transmisión Digital

Capítulo 2. Simulación de la Transmisión Digital

En este capítulo, se hace una descripción del software utilizado para realizar una simulación de una transmisión digital, en esta simulación se mide el desempeño de la transmisión bajo diferentes condiciones de ruido. Además, se hace uso del proceso de codificación convolucional con la decodificación de Viterbi . Esta simulación nos dará elementos para observar el comportamiento en en el desempeño una transmisión utilizando la codificación convolucional con decodificación de Viterbi.

2.1 Descripción de los algoritmos

Los pasos implicados en la simulación de un canal de comunicaciones usando la codificación convolucional y la decodificación de Viterbi son como sigue:

- Generación de los datos, que se transmitirán a través del canal, teniendo como resultado bits de datos binarios
- Codificación Convolucional ,el resultado son símbolos de canal
- Mapeo de los símbolos de canal unos/ceros ,sobre una señal banda base, produciendo símbolos de canal transmitidos
- Agregar ruido a los símbolos de canal transmitido, el resultado son símbolos de canal recibidos.
- Cuantificación de los niveles de canal recibidos, la cuantización con un bit se llama decisión dura, y de dos a n bits de cuantización se llama cuantización suave (n es generalmente tres o cuatro).
- Decodificación de Viterbi en el símbolo de canal cuantizado recibido, resultando otra vez en bits de datos binarios.
- Comparación los bits de datos decodificados a los bits de datos transmitidos y contabilizar el número de errores.

Se puede observar que se omitieron los pasos de la modulación de los símbolos de canal sobre una portadora transmitida, y por lo tanto la demodulación de la portadora recibida para recuperar los símbolos del canal. Sin embargo, podemos modelar exactamente los efectos de AWGN aunque se omitan esos pasos.

2.2 Analisis del listado de programa utilizado para la Simulación

El programa de la simulación abarca una rutina manejadora de la prueba, esta rutina hace el llamado a varias funciones escenciales que realizan las tareas primarias, como la generación datos, la codificación y la decodificación. Estas funciones serán también. Este código simula un enlace a través de un canal AWGN desde la fuente de datos a la salida del decodificador de Viterbi.

2.2.1 Manejador de la Prueba

La simulación de un enlace de comunicaciones con detección y corrección de errores se realiza a través del manejador de prueba. El manejador de prueba primero asigna dinámicamente varios arreglos para almacenar los datos de la fuente, datos a la salida del codificador convolucional, la salida del canal AWGN y de datos a la salida del decodificador de Viterbi. Después, llama las funciones del generador de los datos, introduce los datos generados en el codificador convolucional, simula el ruido de canal, y pasa el resultado al decodificador de Viterbi. Después compara los datos originales del genereador de datos con la salida de datos del decodificador de Viterbi y cuenta el número de errores. Una vez que 100 errores (suficientes para una medida del error del +/- 20% con una confianza del 95%) se acumulen, el conductor de la prueba exhibe la BER para la Es/No dada. Los parámetros de la prueba son controlados por las definiciones en vdsim.h . Por ejemplo, se puede modificar el valor de K para observar el desempeño del decodificador con sus distintos valores o se puede definir o no definir las variables DOENC y DONOENC a conveniencia para realizar las pruebas con codificación o sin codificación respectivamente. A continuación se muestra el contenido de este archivo

```
#define K 3                 /* constraint length o longitud restringida*/
#define TWOTOTHEM 4         /* 2^(K - 1) - cambiar como sea requerido */
#define PI 3.141592654
#define MSG_LEN 1000        /* número de bits en cada mensaje de prueba*/
#define DOENC               /* prueba con codificación
                               convolucional /decodificación de Viterbi */
#undef  DONOENC             /* prueba sin codificación */
#define LOESN0 0.0          /* mínimo Es/No en cada prueba */
#define HIESN0 3.5          /* máximo Es/No en cada prueba */
#define ESN0STEP 0.5        /* incremento de Es/No para el manejador
                               de prueba*/
```

El conductor de la prueba incluye también una opción de compilación para medir la BER para un canal sin codificar, es decir un canal sin la corrección de error. Se utiliza esta opción para validar al generador de ruido gaussiano, comparando la BER simulada sin codificar con la BER sin codificar teórica, dada por $BER = erfc\left(\sqrt{E_b/N_0}\right)/2$ [6], donde E_b/N_0 se expresa como un cociente, no en dB.

A continuación se muestra el listado del código fuente de la rutina que maneja la simulación del enlace. *Junto con el código se han escrito comentarios a modo de documentación* ,que describen las acciones del código correspondiente, posteriormente se hará una descripción detallada de las partes que realizan una labor crítica dentro del programa.

```
/* Manejador de prueba para el decodificador de Viterbi con decisión suave */
/* Version 06.07.2003                                                       */

#include <alloc.h>
#include <conio.h>
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>

#include "vdsim.h"
#include "gen01dat.c"
#include "cnv_encd.c"
#include "addnoise.c"
#include "sdvd.c"

void /*testsdvd*/main(void) {

    long iter, t, long_msg, long_canal; /* variables de bucle, longitudes
                                           del mensaje original y del mensaje
                                           transmitido */
    int *unocer;
    int *codificado;                    /* arreglos para datos originales, codificados y
    int *dvdsout;                       decodificados respectivamente */

    int inicio;

    float *splusn;                      /* arreglos para datos de canal con ruido */

    int i_rxdata, m;                    /* datos rx enteros, m = K - 1*/
    float es_ovr_n0, numero_error_codificado, numero_error_sincod,
          umbral_codificado, umbral_no_codificado, ber_cod, ber_sin_cod; /* varios valores
       estadísticas */

    #if K == 3          /* polinomios para K = 3 */
    int g[2][K] = {(1, 1, 1),      /* 7 */
                   (1, 0, 1));     /* 5 */
    #endif

    #if K == 5          /* polinomios para K = 5 */
    int g[2][K] = {(1, 1,  1, 0, 1),  /* 35 */
                   (1, 0,  0, 1, 1)); /* 23 */
    #endif

    #if K == 7          /* polinomios para K = 7 */
    int g[2][K] = {(1,  1, 1, 1,  0, 0, 1),  /* 171 */
                   (1,  0, 1, 1,  0, 1, 1)); /* 133 */
    #endif

    #if K == 9          /* polinomios para K = 9 */
    int g[2][K] = {(1, 1, 1,  1, 0, 1,  0, 1, 1), /* 753 */
                   (1, 0, 1,  1, 1, 0,  0, 0, 1)); /* 561 */
    #endif

    clrscr();

    printf("\nK = %d", K);

    /* muestra los polimios generadores para el valor de K correspondiente    */
    #if K == 3
    printf("\ng1 = %d%d%d", g[0][0], g[0][1], g[0][2] );
    printf("\ng2 = %d%d%d\n", g[1][0], g[1][1], g[1][2] );
    #endif

    #if K == 5
    printf("\ng1 = %d%d %d%d%d", g[0][0], g[0][1], g[0][2], g[0][3], g[0][4] );
    printf("\ng2 = %d%d %d%d%d\n", g[1][0], g[1][1], g[1][2], g[1][3], g[1][4] );
    #endif

    #if K == 7
    printf("\ng1 = %d %d%d%d %d%d%d", g[0][0], g[0][1], g[0][2], g[0][3], g[0][4],
                    g[0][5], g[0][6] );
    printf("\ng2 = %d %d%d%d %d%d%d\n", g[1][0], g[1][1], g[1][2], g[1][3], g[1][4],
                    g[1][5], g[1][6] );
    #endif

    #if K == 9
    printf("\ng1 = %d%d%d %d%d%d %d%d%d", g[0][0], g[0][1], g[0][2], g[0][3], g[0][4],
```

```
                    g[0][5], g[0][6], g[0][7], g[0][8] );
    printf("\ng2 = %d%d%d %d%d%d %d%d%d\n", g[1][0], g[1][1], g[1][2], g[1][3], g[1][4],
                    g[1][5], g[1][6], g[1][7], g[1][8] );
    #endif

    /* asigna el valor de m */
    m = K - 1;
    /* asigna el valor de la longitud del mensaje declarado en vdsim.h */
    long_msg = MSG_LEN;
    /* calcula y asigna la longitud del mensaje después de pasar a través del canal */
    long_canal = ( long_msg + m ) * 2;

    /* asignar espacio de memoria para los datos del mensaje original no
       codificados */
    unocer = malloc( long_msg * sizeof( int ) );
    if (unocer == NULL) {
       printf("\n testsdvd.c: Error al asignar memoria al arreglo unocer abortando!");
        exit(1);
    }

    /* asignar espacio de memoria para los datos del mensaje codificados */
    codificado = malloc( long_canal * sizeof(int) );
    if (codificado == NULL) {
       printf("\n testsdvd.c:  Error al asignar memoria al arreglo codificado abortando!");
        exit(1);
    }

    /* asignar espacio de memoria para los datos del mensaje codificados con
       ruido sumado */
    splusn = malloc( long_canal * sizeof(float) );
    if (splusn == NULL) {
       printf("\n testsdvd.c:  Error al asignar memoria al arreglo splus abortando!");
       exit(1);
    }

    /* asignar espacio de memoria para para los datos del mensaje decodificados */
    dvdsout = malloc( long_msg * sizeof( int ) );
    if (dvdsout == NULL) {
       printf("\n testsdvd.c:  Error al asignar memoria al arreglo dvdsout abortando!");
        exit(1);
    }

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

    /* Fin de inicialización. Empieza la prueba con un valor de relación señal a ruido
mínimo LOESN0 y que en cada ciclo aumenta una cantidad ESN0STEP hasta un valor máximo HIESN0
los tres valores se encuentran declarados en el archivo vdsim.h. Llama las funciones del
generador de los datos, el codificador convolucional, la simulación del canal, y del
decodificador de Viterbi. Después compara la salida de datos de fuente  con la salida de
datos del decodificador de Viterbi y cuenta el número de errores. Una vez que 100 errores
(suficientes para una medida del error del +/- 20% con una confianza del 95%) se acumulen,
el conductor de la prueba exhibe la BER para la Es/No dada  */

    for (es_ovr_n0 = LOESN0; es_ovr_n0 <= HIESN0; es_ovr_n0 += ESN0STEP) {

        inicio = time(NULL); /* se registra el tiempo para ver la duración de la prueba*/

        /* Inicialización de las variables de número de errores, BER y variable de bucle
        respectivamente, para la prueba con codificación  */
        numero_error_codificado= 0.0;
        ber_cod = 0.0;
        iter = 0;

        /* Si la prueba se realiza con codificación */
        #ifdef DOENC
        /* fija el valor del umbral de números de errores  permitidos en el
           mensaje recibido para la simulación. Una vez que se supere este
           umbral, el conductor de la prueba exhibe la BER para la Es/No dada, con el
        tiempo en que se alcanzó el umbral */
        if (es_ovr_n0 <= 9)
            umbral_codificado = 100; /* +/- 20% */
```

```
else
    umbral_codificado = 20; /* +/- 100 % */

/* Aquí es momento en que se realiza la generación y procesamiento de los datos */
while (numero_error_codificado< umbral_codificado) {
    iter += 1;

    /*printf("Generando los datos uno-cero \n");*/
    gen01dat(long_msg, unocer);

    /*printf("codificando convolucionalmente los datos\n");*/
    cnv_encd(g, long_msg, unocer, codificado);

    /*printf("Sumando ruido a los datos codificados \n");*/
    addnoise(es_ovr_n0, long_canal, codificado, splusn);


    /*printf("Decodificando los datos del canal BSC data\n");*/
    sdvd(g, es_ovr_n0, long_canal, splusn, dvdsout);

    /* cuenta el número de errores */
    for (t = 0; t < long_msg; t++) {
        if ( *(unocer + t) != *(dvdsout + t) ) {
            /*printf("\n error ocurrido en la locación %ld", t);*/
                numero_error_codificado+= 1;
        } /* end if */
    } /* end t for-loop */


   /* interrumpe la prueba si el usuario lo solicita */
   if (kbhit()) exit(0);
    /*printf("\nRealizado!");*/

} /* fin del loop-while */

/*Se calcula la tasa de bit en error para el mensaje transmitido con codificación */
ber_cod = numero_error_codificado/ (long_msg * iter);

/* calcula el tiempo transcurrido para la prueba con un es_ovr_n0 dado*/
printf("\nEl tiempo de término fue %d segundos para %d iterations",
          time(NULL) - inicio, iter);
#endif

 /* Inicialización de las variables de número de errores, BER y variable de bucle
respectivamente, para la prueba sin codificación  */

numero_error_sincod = 0.0;
ber_sin_cod = 0.0;
iter = 0;

/* Si la prueba se realiza sin codificación */
#ifdef DONOENC
/* fija el valor del umbral de números de errores  permitidos en el
   mensaje recibido sin codificación para la simulación. Una vez que se supere este
   umbral, el conductor de la prueba exhibe la BER para la Es/No dada */
if (es_ovr_n0 <= 12)
    umbral_no_codificado = 100;
else
    umbral_no_codificado = 20;


while (numero_error_sincod < umbral_no_codificado) {
    iter += 1;

/* Aquí es momento en que se realiza la generación y procesamiento de los datos */
/* Se observa que se realiza la transmisión sin codificación ni decodificación */
    /* printf("Generando los datos uno-cero \n");*/
    gen01dat(long_msg, unocer);

    /* printf("Sumando ruido a los datos no codificados \n");*/
    addnoise(es_ovr_n0, long_msg, unocer, splusn);
```

```
            /* hace una decisión dura sobre los símbolos recibidos y cuenta el
               número de errores en la secuencia recibida */
            for (t = 0; t < long_msg; t++) {

                if ( *(splusn + t) < 0.0 )
                    i_rxdata = 1;
                else
                    i_rxdata = 0;

                if ( *(unocer + t) != i_rxdata )
                    numero_error_sincod += 1;
            }

            /* interrumpe la prueba si el usuario lo solicita */
            if (kbhit()) exit(0);
            /*printf("\nRealizado!");*/

        }/* fin del loop-while */

        /* calcula la tasa de bit en error para el mensaje transmitido sin codificación */
        ber_sin_cod = numero_error_sincod / (long_msg * iter);
        #endif

        printf("\nEn %1.1fdB Es/No, ", es_ovr_n0);

        /* Se muestran
        #ifdef DOENC
        printf("la ber con codificación fue %1.1e ", ber_cod);
        #ifdef DONOENC
        printf(" y ");
        #endif
        #endif

        #ifdef DONOENC
        printf("la ber sin codificación fue %1.1e", ber_sin_cod);
        #endif


    }

    /* liberar la memoria reservada dinámicamente para los arreglo */
    free(unocer);
    free(codificado);
    free(splusn);
    free(dvdsout);

    while ( !kbhit() ) {
    }

    exit(0);
}
```

A continuación se hace una descripción de los módulos correspondientes a la generación de datos, codificación convolucional, simulador del canal y decodificador que se utilizaron en la simulación. Cabe señalar que *los módulos de codificación y decodificación son básicamente los mismos módulos que se utilizarán en la implementación de la transmisión, con algunas variantes.*

2.2.2 Generador de datos

La función del generador de datos es simular la fuente de datos. Acepta como argumento el número de bits a generar y un puntero a un arreglo de entrada, y llena el arreglo con ceros y unos elegidos al azar.

La generación de los datos que se transmitirán a través del canal se puede lograr simplemente usando un generador de números al azar. Uno que produce una distribución uniforme de números en el intervalo 0 a un valor máximo se proporciona en C: `rand ()` . La función randomize inicia el generador de números aleatorios a un valor aleatorio, usando la función time( ), por lo que es necesario incluir time.h

Usando esta función, podemos decir que cualquier valor menor que la mitad del valor máximo es un cero; cualquier valor mayor que o el igual a la mitad del valor máximo es un uno.

```
/* GENERADOR  DE DATOS (0/1)                                          */
/* Version 06.07.2003                                                 */
 #include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
 #include "vdsim.h"

void gen01dat( long long_datos, int *arreglo_sal ) {
     long t;            /* tiempo */
    randomize();
    /* genera datos aleatorios y escribe estos en el arreglo de salida */
    for (t = 0; t < long_datos; t++)
        *( arreglo_sal + t ) = (int)( rand() / (RAND_MAX / 2) > 0.5 );
}
```

Como se observa en el código, en el arreglo **arreglo_sal** se guarda la información generada de forma aleatoria, a la cual se le se aplicara la codificación, se le agregará ruido y finalmente será decodificada. Podemos observar también, que para obtener solo valores 1 y 0 se toma el valor generado por rand( ) y lo dividimos entre la mitad del valor máximo, dado por RAND_MAX, esto es:

$$\text{rand()} / (\text{RAND\_MAX} / 2)$$

Con esta operación nos aseguramos de que el resultado de la operación sea mayor o igual a uno, cuando rand( ) es mayor o igual a la mitad de RAND_MAX . Luego aplicamos el operador relacional '>'. Las expresiones que utilizan los operadores relacionales y lógicos devuelven 1 en caso de cierto y 0 en caso de falso, por lo que se concluye que cuando el resultado de la división es mayor o igual a 1 el valor guardado en el arreglo en uno y cuando el resultado de la división es menor a 1 el valor guardado es cero

2.2.3 Codificación Convolucional de los datos

La función del codificador convolucional acepta como argumentos el número de bits en el arreglo de entrada y punteros a los arreglos de entrada y de salida. Después realiza la codificación convolucional especificada y llena el arreglo de unos/ceros de símbolos de canal de salida. Los parámetros del código convolucional están en el archivo cabecera vdsim.h.

Para entender la implementación que se hizo del codificador, veamos cual es su funcionamiento antes de explicar el código que lo implementa La codificación convolucional de los datos se logra usando un registro de desplazamiento y una lógica combinatoria asociada que realice la suma modulo-dos, el registro de desplazamiento podría ser propuesto simplemente como una cadena de flip-flops en donde la salida del n-ésimo flip-flop se envía a la entrada del (n+1)-ésimo flip-flop . Cada vez que el flanco activo del reloj ocurre, la entrada al flip-flop se registra en la salida, y los datos son desplazados así a otra etapa.) La lógica combinatoria está a menudo en la forma de conexión en cascada de compuertas or-exclusivo. Cabe recordar, que las compuertas or-exclusivo tienen dos entradas una salida, y son representadas a menudo por el símbolo de la lógica demostrado abajo, este implementa la tabla de verdad siguiente:

Entrada A	Entrada B	Salida (A xor B)
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 2.1 Tabla de verdad de la compuerta or-exclusiva, que sirve para implementar la suma modulo-dos

Figura 2.1 Compuerta or-exclusiva, que sirven a menudo para la implementación de la codificación convolucional.

La compuerta or-exclusivo realiza la adición del modulo-dos de sus entradas. Cuando se conectan en cascada q compuertas de dos entradas or-exclusivo, con la salida de la primera alimentando una de las entradas de la segunda, la salida de segunda alimentando una de las entradas de tercera, el etc., la salida de la última en la cadena es la suma módulo-dos de las q + 1 entradas.

Otra manera de ilustrar el sumador modulo-dos, y la manera que se utiliza comúnmente en libros de textos, es un círculo con un símbolo + adentro, así:

Figura 2.2 Representación de la suma modulo-dos

Ahora que tenemos los dos componentes básicos del codificador convolucional (flip-flop que abarcan el registro de desplazamiento y las compuertas or-exclusivo que abarcan los sumadores modulo-dos) definidos, vamos a mirar una implementación de un codificador convolucional para un código de tasa 1/2, K = 3, m = 2:

En este codificador, los bits de datos se proporcionan a una tasa de k bits por segundo. Los símbolos del canal se hacen salir a una tasa de n = 2k símbolos por segundo. El bit de la entrada es estable durante el ciclo del codificador. El ciclo del codificador comienza cuando ocurre un flanco o borde de reloj de la entrada. Cuando ocurre el borde de reloj de la entrada, la salida del flip-flop izquierdo se registra en el flip-flop derecho, el bit anterior de entrada se registra en el flip-flop izquierdo, y un nuevo bit de la entrada llega a estar disponible. Entonces las salidas de los sumadores superior e inferior modulo-dos llegan a ser estables.

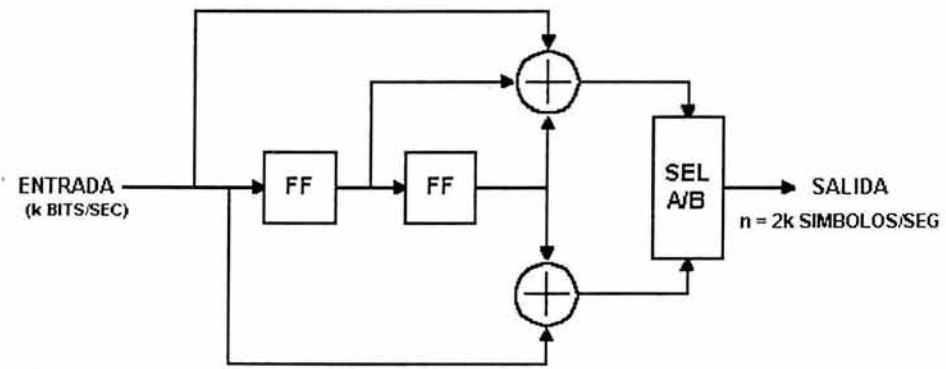


Figura 2.3 Implementación de un codificador codificador convolucional para un código de tasa 1/2, K = 3, m = 2

El selector de la salida (bloque SEL A/B) completa un ciclo con dos estado, en el primer estado, él selecciona y hace salir la salida del sumador modulo-dos superior ; en el segundo estado, selecciona y hace salir la salida del sumador del modulo-dos inferior.

El codificador mostrado arriba codifica un código convolucional K = 3, (7, 5). Los números octales 7 y 5 representan los generadores polinomiales de código, que cuando se leen en binario (111_2 y 101_2) corresponde a las conexiones del registro de desplazamiento a los sumadores modulo-dos superior e inferior, respectivamente. Este código se ha determinado para ser el "mejor" código para tasa 1/2, K = 3. Es el código que utilizaré para la discusión y los ejemplos restantes, por las razones que llegarán a ser fácilmente evidentes cuando nos introduzcamos en el algoritmo del decodificador de Viterbi.

Vamos mirar un ejemplo de secuencia de datos de entrada, y la secuencia de datos correspondiente de salida:

Sea la siguiente secuencia de entrada 0101110010100012.

Asuma que las salidas de ambos flip-flop en el registro de desplazamiento son limpiados inicialmente, es decir sus salidas son ceros. El primer ciclo de reloj hace que el primer bit de entrada, un cero, sea disponible para el codificador. Las salidas del flip-flop son ambos ceros. Las entradas a los sumadores modulo-dos son todos los ceros, así que la salida del codificador es 00_2.

El segundo ciclo de reloj hace que el segundo bit de entrada este disponible para el codificador. El flip-flop izquierdo registra el bit anterior, y el flip-flop derecho registra la a salida cero de el flip-flop izquierdo. Las entradas al sumador modulo-dos superior son 100_2, así que la salida es 1. Las entradas al sumador modulo-dos inferior son 10_2, así que la salida es también 1. El codificador hace salir 11_2 para los símbolos de canal.

El tercer ciclo de reloj hace que el tercer bit de entrada, un cero, disponible para el codificador. En el flip-flop izquierdo registra el bit anterior, que era uno, y el flip-flop derecho registra un cero de hace dos tiempos de bit atrás. Las entradas al sumador módulo-dos superior son 010_2, así que la salida es uno. Las entradas al sumador módulo-dos inferior son 00_2, así que la salida es cero. El codificador hace salir 10_2 para los símbolos de canal, etcétera. El diagrama de tiempo mostrado abajo ilustra el proceso:

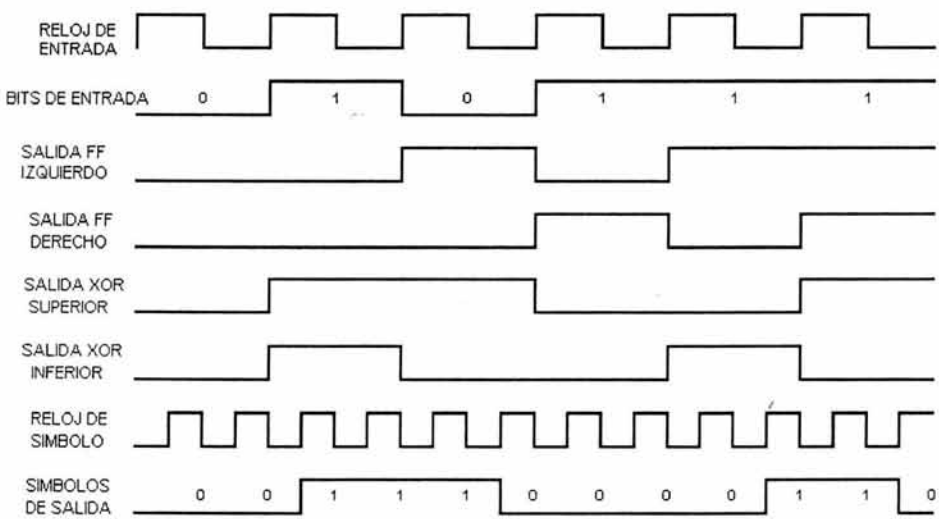


Figura 2.4 Diagrama de tiempo del codificador de una secuencia de datos de entrada, y la secuencia de datos correspondiente de salida:

después de que todas las entradas se hayan presentado al codificador, la secuencia de la salida será:

$00\ 11\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 10\ 00\ 10\ 11\ 00\ 11_2$.

Nótese que se han formado pares con las salidas del codificador, el primer que BIT de cada par son la salida del sumador módulo-dos superior; el segundo bit en cada par es la salida del sumador módulo-dos inferior.

Se puede ver de la estructura del codificador convolucional de razón ½ K = 3 y del ejemplo dado anteriormente que cada bit de entrada tiene un efecto en tres pares sucesivos de símbolos de salida. Eso es un punto extremadamente importante y esto es lo que da al código convolucional su poder de corrección de error. La razón llegará a ser evidente cuando nos adentremos en el algoritmo decodificador de Viterbi.

Ahora si vamos solamente a enviar los 15 bits de datos dados arriba, y si el bit último afecta tres pares de símbolos de la salida, necesitamos hacer salir dos pares más de símbolos. Esto es logrado en nuestro codificador del ejemplo haciendo registrar los flip-flops del codificador convolucional dos (= m) veces más, mientras mantengamos la entrada en cero. Esto se llama "limpiar" [1] el codificador, y resulta en dos pares más de símbolos de la salida. La salida binaria final del codificador es entonces 00 11 10 00 01 10 01 11 11 10 00 10 11 00 11 10 11_2. Si no realizamos la operación que limpia, los últimos m bits del mensaje tienen menos capacidad de corrección de error que los primeros hasta los m-1 bits tenían. Esto es una cosa importante para recordar si se va a utilizar esta técnica de FEC en un ambiente de envíos de ráfagas. El paso de despejar el registro de desplazamiento al principio de cada ráfaga se debe tomar en cuenta. El codificador debe comenzar en un estado conocido y terminar en un estado sabido para que el decodificador pueda reconstruir la secuencia de los datos de entrada correctamente.

Representemos al codificador como máquina simple de estados. El codificador del ejemplo tiene dos bits de memoria, entonces tiene cuatro estados posibles. Vamos a dar al flip-flop izquierdo al peso binario de 2^1, y al flip-flop derecho un peso binario de 2^0. Inicialmente, el codificador está en el estado de todos ceros. Si el primer bit de entrada es un cero, el codificador permanece en el estado de todo ceros en el flanco de reloj siguiente. Pero si el bit de la entrada es uno, las transición del codificador es al estado 10_2 en el flaco de reloj siguiente. Entonces, si el bit siguiente de entrada es cero, la transiciones del codificador es al estado 01_2, de otra manera, m la transición es al estado 11_2. La tabla siguiente da el estado siguiente, dado el estado actual y la entrada, con los estados dados en binario:

	Estado siguiente, si	
Estado Actual	**Entrada = 0:**	**Entrada = 1:**
00	00	10
01	00	10
10	01	11
11	01	11

Tabla 2.2 Tabla de transición de estados para el codificador del ejemplo

[1] El término utilizado en inglés es flushing.

La tabla anterior a menudo se le llama una tabla de transición de estados. Le llamaremos como la `tabla del siguiente del estado`. Ahora miremos una tabla que enumere los símbolos de canal de salida, dados el estado actual y los datos de entrada, la cual nos referiremos como `la tabla de salida`:

	Símbolos de la salida, si	
Estado Actual	**Entrada = 0:**	**Entrada = 1:**
00	00	11
01	11	00
10	10	01
11	01	10

Tabla 2.3 Tabla de salida para el codificador del ejemplo

Se debe ahora ver, que con estas dos tablas se puede describir totalmente el comportamiento del codificador convolucional del ejemplo de tasa ½, K = 3. Observe que ambas tablas tienen $2^{(K-1)}$ los renglones, y 2^{k} columnas, donde K es la longitud restringida y k es el número de bits que entraron al codificador para cada ciclo. Estas dos tablas serán manejadas cuando comenzamos a discutir el algoritmo decodificador de Viterbi.

Veamos ahora la parte del código que implementa a este codificador convolucional,.

```
/* CODIFICADOR CONVOLUCIONAL                                          */
/* Version Version 06.07.2003                                         */

#include <alloc.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>

#include "vdsim.h"

/* La función del codificador convolucional  acepta como argumento  la matriz g[2][K] que
representa los generadores polinomiales de código, que cuando se leen en binario (111₂  y
101₂) corresponde a las conexiones del registro de desplazamiento a los sumadores modulo-dos
superior e inferior, respectivamente; punteros a los arreglos de  entrada y de salida y el
número de bits  en el arreglo  de entrada. */

void cnv_encd(int g[2][K], long long_ent, int *arreglo_in, int *arreglo_out) {

    int m;                      /* K - 1 */
    long t, tt;                 /* tiempo de bit, tiempo de simbolo */
    int j, k;                   /* variables de loop  */
    int *unencoded_data;        /* apuntador a los arreglos de dato */
    int shift_reg[K];           /* registro de desplazamiento del codificador */
    int sr_head;                /* indice a la primera etapa del registro de desplazamiento
*/
    int p, q;                   /* Salidas de las compuertas XOR superior e inferior */
```

```
    m = K - 1;

    /* asignar espacio de memoria para para los datos no codificados del arreglo de datos de
entrada*/
    unencoded_data = malloc( (long_ent + m) * sizeof(int) );
    if (unencoded_data == NULL) {
        printf("\ncnv_encd.c: No puedo asignar suficiente espacio de memoria para los datos
no codificados!    Abortando....");
        exit(1);
    }

    /* leer los datos de entrada y almacenar en el arreglo de datos no codificados */
    for (t = 0; t < long_ent; t++)
        *(unencoded_data + t) = *(arreglo_in + t);

    /* asignar m ceros en el final  de los datos */
    for (t = 0; t < m; t++) {
        *(unencoded_data + long_ent + t) = 0;
    }

    /* Inicializar el registro de desplazamineto */
    for (j = 0; j < K; j++) {
        shift_reg[j] = 0;
    }
    /* Para tratar de mejorar un poco la velocidad de ejecución, el registro de
desplazamiento  debe operar como un bufer circular, entonces se necesitará
al menos un apuntador de cabecera. No se necesitará un apuntador de  cola,
solo sobreescribiremos la entrada más antigua con la entrada más nueva */


            /* Inicializar el indice indice de símbolo de canal de salida */
    tt = 0;

    /* Ahora iniciar el proceso de codificación */
    /* Computar las salidas p y q del sumador módulo dos superior e inferior,
    un bit a la vez, cada ciclo del for procede a procesar un bit del mensaje */
    for (t = 0; t < long_ent + m; t++) {
    /* Entra un dato en el registro de desplazamiento */
        shift_reg[sr_head] = *( unencoded_data + t );
        p = 0;
        q = 0;
        for (j = 0; j < K; j++) {
    /* Donde k es el índice del registro de desplazamiento que indica el orden en que se
debe aplicar la lógica binaria(operaciones AND y XOR )en los bits de datos que se encuentra
en el registro de desplazamiento con la matriz g que contiene los coeficientes del polinomio
generador */
            k = (j + sr_head) % K;
    /* aquí se realiza, primero mediante la operación AND, la elección de qué bits van ser
utilizados por la compuerta or-exclusivo que realiza la adición módulo-dos y que en lenguaje
C se realiza con el el operador ^. Donde p y q so  las salidas de las compuertas XOR
superior e inferior respectivamente */

            p ^= shift_reg[k] & g[0][j];
            q ^= shift_reg[k] & g[1][j];
        }

        /* escribir las salidas de las compuertas XOR superior e inferior
       como símbolos de canal */
        *(arreglo_out + tt) = p;
        tt = tt + 1;
        *(arreglo_out + tt) = q;
        tt = tt + 1;


        sr_head -= 1;     /* equivalente a hacer un corrimiento a la derecha de
                    todo un lugar  */
        if (sr_head < 0)
            sr_head = m;

    }
```

```
    /* Liberar dinamicamente la memoria asignada  al arreglo*/
    free(unencoded_data);

}
```

La parte inicial del código, no es más que la iniciación de los arreglos que almacenan los datos, así tenemos que arreglo para datos no codificados (unencoded_data) que primero se le reserva espacio de memoria, se le asignan los datos a codificar que vienen del arreglo que se recibe como parámetro de entrada del procedimiento (arreglo_in) y que además se le agregan m bits ceros de para limpiar el registro. Posteriormente se pone en ceros el registro de desplazamiento (shift_reg), este el registro que, como se propuso anteriormente, podría ser implementa con flip-flop's y para nuestro programa es implementado con un simple arreglo.

Como se menciona en los comentarios del código, para tratar de mejorar un poco la velocidad de ejecución, el registro de desplazamiento debe operar como un bufer circular, esto es, no perderemos tiempo en borrar los datos presentes en el registro y volverlos a escribir en su nuevo lugar con la finalidad de ser desplazados un lugar. Con la programación implementada, solo sobrescribiremos la entrada más antigua con la entrada más nueva, entonces se necesitará al menos un apuntador de cabecera, este apuntador de cabecera es la variable *sr_head* que guarda la posición en el registro de desplazamiento en donde está el dato más nuevo.

Una vez entrado el registro más nuevo se procede a aplicar la lógica combinatoria, que como se mencionó con anterioridad por lo regular esta implementada como una cascada de compuertas or-exclusivo, pero que la manera en que se programó fue con un ciclo que opera bit a bit los elementos del registro de desplazamiento y aplicando los operadores AND y XOR. Considere el siguiente ejemplo:

Sean las matrices *g* y *unencoded_data* siguientes:

$$g=\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad \text{unencaded\_data} = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix}$$

Entonces, para entender como funciona el proceso de codificación vamos a revisar detalladamente el ciclo. Para un `codificador convolucional de razón ½ K = 3`, tenemos en el inicio, el registro de desplazamiento, implementado con el arreglo *shift_reg* con cero en sus tres elementos

shift_reg=

(0)	(1)	(2)
0	0	0

y los valores p=0 y q=0 que son las salidas binarias del codificador.

Entonces sr_head es el puntero que indica el elemento más nuevo en el registro de desplazamiento y k (que es calculado mediante el operador división en módulo '%') es un puntero que indica qué elemento del registro de desplazamiento vamos a procesar, tt es un puntero que indica que elemento de la matriz de salida vamos a regresar. Siguiendo la parte

del código a partir del momento donde se indica que inicia el proceso de codificación, tenemos lo siguiente:
Para el lazo más externo
sr_head=0 ; t=0
shift_reg(0)=unencode_data(0); el registro de desplazamiento toma el primer elemento no codificado, esto es

shift_reg=

(sr_head)	(1)	(2)
1	0	0

para el lazo interno, donde se aplica la lógica binaria, bit por bit, y el cálculo de las salidas binarias del codificador, tenemos

k = (0+0)%3 = 0 ;
p=p XOR (shift_reg(0) AND g(0,0))=0 XOR (1 AND 1) =1
q=q XOR (shift_reg(0) AND g(1,0))=0 XOR (1 AND 1) =1

siguiente ciclo

k=(1+0)%3=1 ;
p=p XOR (shift_reg(1) AND g(0,1))=1 XOR (0 AND 1) =1
q=q XOR (shift_reg(1) AND g(1,1))=1 XOR (0 AND 0) =1

siguiente ciclo

k=(2+0)%3=2 ;
p=p XOR (shift_reg(2) AND g(0,2))=1 XOR (0 AND 1) =1
q=q XOR (shift_reg(2) AND g(1,2))=1 XOR (0 AND 1) =1

fin del lazo interno, por lo que el registro de salida queda
arreglo_out(0)=p=1
arreglo_out(1)=q=1

se inicia otro ciclo, por lo que ahora, según las líneas de código, tenemos los siguientes valores
sr_head=m=2 ; t=1
shift_reg(2)=unencode_data(1); el registro de desplazamiento toma el segundo elemento no codificado, esto es

shift_reg=

(0)	(1)	(sr_head)
1	0	0

note que ahora en elemento más nuevo es shift_reg(2) y *no se tuvo que desplazar de su lugar ningún otro elemento.* Además se pone p=0 y q=0.

Para el lazo donde se aplica la lógica binaria y el cálculo de las salidas binarias del codificador, tenemos
k=(0+2)%3=2 ;

p=p XOR (shift_reg(2) AND g(0,0))=0 XOR (0 AND 1) =0
q=q XOR (shift_reg(2) AND g(1,0))=0 XOR (0 AND 1) =0

siguiente ciclo

k=(1+2)%3=0 ;
p=p XOR (shift_reg(0) AND g(0,1))=0 XOR (1 AND 1) =1
q=q XOR (shift_reg(0) AND g(1,1))=0 XOR (1 AND 0) =0

siguiente ciclo

k=(2+2)%3=1 ;
p=p XOR (shift_reg(1) AND g(0,2))=1 XOR (0 AND 1) =1
q=q XOR (shift_reg(1) AND g(1,2))=0 XOR (0 AND 1) =0

fin del lazo interno, por lo que el registro de salida queda
arreglo_out(2)=p=1
arreglo_out(3)=q=0

se inicia otro ciclo, por lo que ahora, según las líneas de código, tenemos los siguientes valores
sr_head= sr_head -1=1 ; t=2
shift_reg(1)=unencode_data(2); el registro de desplazamiento toma el tercer elemento no codificado, esto es

shift_reg=

(0)	(sr_head)	(2)
1	1	0

note que ahora en elemento más nuevo es shift_reg(1) y nuevamente, *no se tuvo que desplazar de su lugar ningún otro elemento.* Además se pone p=0 y q=0.

Para el lazo donde se aplica la lógica binaria y el calculo de las salidas binarias del codificador, tenemos
k=(0+1)%3=1 ;
p=p XOR (shift_reg(1) AND g(0,0))=1 XOR (0 AND 1) =1
q=q XOR (shift_reg(1) AND g(1,0))=1 XOR (0 AND 1) =1

siguiente ciclo

k=(1+1)%3=2 ;
p=p XOR (shift_reg(2) AND g(0,1))=1 XOR (0 AND 1) =1
q=q XOR (shift_reg(2) AND g(1,1))=1 XOR (0 AND 0) =1

siguiente ciclo

k=(2+1)%3=0 ;
p=p XOR (shift_reg(0) AND g(0,2))=1 XOR (1 AND 1) =0

q=q XOR (shift_reg(0) AND g(1,2))=1 XOR (1 AND 1) =0

fin del lazo interno, por lo que el registro de salida queda
arreglo_out(2)=p=0
arreglo_out(3)=q=0

nuevamente note lo que pasa con k, k va tomando valores de tal forma que nos indica los elementos del registro de desplazamiento, desde el más nuevo hasta el más viejo. Las siguientes iteraciones son muy similares, solo falta introducir el último elemento del arreglo de los datos no codificados y los dos bits cero de limpieza para terminar el total de ciclos.

Finalmente el arreglo de salida quedaría:

Arreglo_out=

(0)	(1)	(2)	(3)	(4)	(5)
1l	1	1	0	0	0

2.2.4 Simulador de Canal BPSK

La función de la simulación del canal acepta como argumentos la E s /N 0 deseada, el número de símbolos de canal en el arreglo de entrada, y punteros a los arreglos de entrada y de la salida. Realiza el mapeo de binario (uno y cero) a niveles de señal de banda base (+/- 1) en los símbolos de canal de la salida del codificador convolucional. Entonces agrega variables aleatorias gaussianas a los símbolos mapeados, y llena el arreglo de la salida. Los datos de salida son números de punto flotante. A continuación se muestra el código fuente de este módulo

```
/* SIMULADOR DE CANAL BINARIO SIMÉTRICO                                    */
/* Version Version 06.07.2003                                                */

#include <alloc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "vdsim.h"


/* Función que genera números aleatorios con distribución gaussiana para simular el ruido
sumado en el canal durante la transmisión de la señal. */

float gngauss(float media, float sigma);

/* Inicio del módulo que trasforma de mensaje binario a símbolos de canal y agrega
ruido gaussiano a los símbolos transmitidos*/

void addnoise(float es_ovr_n0, long long_canal, int *arreglo_in, float *arreglo_out) {
```

```
    long t;
    float media, es, razon_sn, sigma, simbolo;

    /*dada la Es/No deseada (para BPSK = Es/No - 3 dB), calcula la desviaciòn
    estandar de el ruido gaussiano aditivo blanco (AWGN). La desviación estandar
    de el AWGN debe ser usada para generara variables aleatorias gaussianas
    simulando el ruido que es sumado a la señal. */


    media = 0;
    es = 1;
    razon_sn = (float) pow(10, ( es_ovr_n0 / 10) );
    sigma =  (float) sqrt (es / ( 2 * razon_sn ) );

    /* ahora transformar los datos 0/1 a +1/-1 y sumar ruido */
    for (t = 0; t < long_canal; t++) {

       /* Si el valor del dato es 1, el simbolo de canal es -1; si el valor del dato
       binario es 0, el símbolo de canal es +1*/

        simbolo = 1 - 2 * *( arreglo_in + t );

        /*  Ahora generar el ruido gaussiano, sumar este al símbolo de canal y
          producir la salida del símbolo de canal con ruido  */

        *( arreglo_out + t ) = simbolo + gngauss(media, sigma);
    }

}

/* función para generar números con distribución gaussiana para simular el ruido */
float gngauss(float media, float sigma) {

    /* Esta función utiliza el hecho de que una variable aleatoria R con ditribución
    de Rayleigh, con la distribución de probabilidad F(R) = 0 si R < 0 y F(R) =
    1 - exp(-R^2/2*sigma^2) si R >= 0, es relacionada a un par de variables
    Gaussianas C y D a través de la transformación C = R * cos(theta) y
    D = R * sin(theta), donde theta es ima variable uniformemente distribuida
    en el intervalo (0, 2*pi()). */

    double u, r;            /* variables aleatorias uniforme y de Rayleigh */

    /* genera un número aleatorio uniformemente distribuido u entre 1 y 1E-6*/
    u = (double)rand() / RAND_MAX;
    if (u == 1.0) u = 0.999999999;

    /* genera un número aleatorio con distribución de Rayleigh r usando u */
    r = sigma * sqrt( 2.0 * log( 1.0 / (1.0 - u) ) );

    /* genera otro número aleatorio uniformemente distribuido u como el anterior */
    u = (double)rand() / RAND_MAX;
    if (u == 1.0) u = 0.999999999;

    /* genera y retorna un número aleatorio Gaussiano usando r y u */
    return( (float) ( media + r * cos(2 * PI * u) ) );
}
```

2.2.4.1 Mapeo de los símbolos de canal en niveles de señal

El Mapeo de los unos/ceros de salida del codificador convolucional sobre un esquema de señalización en banda base bipolar es simplemente una cuestión de traducir ceros a +1's y a unos a –1's. Esto puede ser logrado realizando la operación y = 1 - 2x en cada símbolo de salida del codificador convolucional.. El segmento de código que realiza

esta función, perteneciente al módulo Simulador de Canal Binario Simétrico, se muestra a continuación:

```
/* Si el valor del dato es 1, el simbolo de canal es -1; si el valor del dato
binario es 0, el símbolo de canal es +1*/

simbolo = 1 - 2 * *( arreglo_in + t );
```

2.2.4.2Adición de ruido a los símbolos transmitidos

La adición de ruido a los símbolos transmitidos de canal producido por el codificador convolucional implica generar números aleatorios gaussianos, escalando los números según la energía deseada por símbolo a la razón de densidad de ruido, E_s/N_0, y sumando los números aleatorios gaussianos escalados a los valores de símbolo de canal. El segmento de código que realiza esta función, perteneciente al módulo Simulador de Canal Binario Simétrico, se muestra a continuación:

```
for (t = 0; t < long_canal; t++) {

    *( arreglo_out + t ) = simbolo + gngauss(mean, sigma);
}
```

Para el canal sin codificar, $E_s/N_0 = E_b/N_0$, puesto que hay un símbolo del canal por bit. Sin embargo, para el canal cifrado, $E_s/N_0 = E_b/N_0 + 10\log_{10}(k/n)$[6]. Por ejemplo, para la tasa de codificación 1/2,

$$E_s/N_0 = E_b/N_0 + 10\log_{10}(1/2) = E_b/N_0 - 3.01 \text{ dB}.$$

Similarmente, para la tasa de codificación 2/3,

$$E_s/N_0 = E_b/N_0 + 10\log_{10}(2/3) = E_b/N_0 - 1.76 \text{ dB}.$$

El generador de números aleatorios gaussianos es la única parte interesante de esta tarea. C proporciona solamente un generador de números aleatorios con distribución uniforme, `rand()` . Para obtener números aleatorios gaussianos, nos aprovechamos de relaciones entre las distribuciones uniforme, de Rayleigh, y la gaussiana:
Dado una variable aleatoria uniforme U, una variable aleatoria de Rayleigh R se puede obtener por[6]:

$$R = \sqrt{2\cdot\sigma^2\cdot\ln(1/(1-U))} = \sigma\cdot\sqrt{2\cdot\ln(1/(1-U))}$$

donde σ^2 es la varianza de la variable aleatoria de Rayleigh, y dado R con una segunda variable aleatoria uniforme V, dos variables aleatorias gaussianas G y H se pueden obtener por

$$G = R\cos(V) \text{ y } H = R\sin(V).$$

En el canal AWGN , la señal es corrompida por ruido aditivo, n(t), el cual tiene el potencia espectral *No*/2 watts/Hz. La varianza σ^2 de este ruido es igual a *No*/2 . Si ajustamos la energía por símbolo E_s igual a 1, entonces $E_s/N_0 = 1/2\sigma^2$. De tal manera que $\sigma = \sqrt{1/(2\cdot(E_s/N_0))}$ [6].

La función que realiza la generación de números aleatorios para simular el ruido gaussiano , acepta como parámetros la media del símbolo y σ, se encuentra en el módulo de simulación del canal binario mostrado anteriormente y es la siguiente

```
/* función para generar números con distribución gaussiana para simular el ruido */
float gngauss(float media, float sigma)
```

2.2.5 Ejecución Del Decodificador De Viterbi

El decodificador de Viterbi en sí mismo es el foco primario de este tema en particular. Los argumentos de la función del decodificador de Viterbi son la E_s/N_0 prevista el número de símbolos de canal en el arreglo de la entrada, y los punteros a los arreglos de la entrada y de la salida. Primero, la función del decodificador instala sus estructuras de datos. Entonces, realiza la cuantización suave de tres-bits en los símbolos recibidos de punto flotante del canal, usando la Es/No, prevista, produciendo números enteros. (opcionalmente, un cuantizador fijo diseñado para 4 dB de Es /N 0 se puede elegir.) Esto termina el proceso preliminar.

El paso siguiente es comenzar a descifrar los símbolos del canal de decisión suave. El decodificador construye un enrejado de profundidad K x 5, y después remonta de nuevo al principio del enrejado y da salida a un bit. El decodificador entonces desplaza el enrejado a la izquierda un instante del tiempo, desechando los datos más viejos, a continuación computa la métrica acumulada del error para el próximo instante de tiempo, remonta de nuevo al principio del enrejado y da salida a un bit. El decodificador continúa de esta manera hasta que alcanza los bits de limpieza. Los bits de limpieza hacen que el codificador converja de nuevo al estado 0, y el decodificador explota este hecho. Una vez que el decodificador construya el enrejado para el último bit, limpia el enrejado, descifrando y haciendo salir todos los bits en el enrejado pero no incluyendo los primeros bits de limpieza. Para consultar el código completo del decodificador de viterbi, refierase al disco anexo a este decumento.

A continuación se hace una descripción de los módulos correspondientes al decodificador de Viterbi que se utilizaron en la simulación y que se utilizarán en la implementación de la transmisión, con algunas variantes.

2.2.5.1 Cuantización de los símbolos de canal recibidos

Un decodificador ideal de Viterbi debe trabajar con una precisión infinita, o al menos con números de punto flotante. En sistemas prácticos, se cuantizan los símbolos de canal recibidos con uno o unos pocos bits de precisión para reducir la complejidad del decodificador de Viterbi. Si los símbolos de canal recibidos son cuantizados con más de un bit de precisión, el resultado es llamado datos con decisión suave. Un decodificador con datos de entrada con decisión suave cuantizados con tres o cuatro bits de precisión puede desempeñarse mejor con alrededor de 2 dB que uno que este trabajando con entradas de decisión dura. La precisión de cuantización usual es tres bits. Más bits presenta un poco de mejora adicional.

La selección de los niveles de cuantización son una importante decisión en el diseño ya que esto puede tener un efecto significativo en el desempeño del enlace. La siguiente es una breve explicación de una manera de seleccionar los niveles. Asuma que nuestros niveles de señal recibida en ausencia de ruido son -1V = 1, +1V = 0. Con ruido, nuestra señal tiene media +/- 1 y desviación estandar $\sigma = \sqrt{1/(2\cdot(E_s/N_0))}$. Usemos un cuantizador uniforme con tres bits teniendo las relaciones de entrada/salida mostradas en la figura 2.5, donde D es un nivel de decisión que se debe calcular.

La parte del programa encargada de la cuantización se presenta a continuación. Esta parte inicializa un cuantizador de decisión suave de tres bits para una Eb/No variable expresada en dB que se recibe como parámetro. Aquí se crea un arreglo que servirá de referencia para mapear el valor de punto flotante del símbolo de canal recibido con ruido a un valor entero. Se dice que es un cuantizador de tres bits, porque hay 2^m=8 niveles de cuantización, del 0 al 7, que corresponde a un valor de m=3, donde m es el número de bits que representa el número de bits que se usarían en cada muestra.

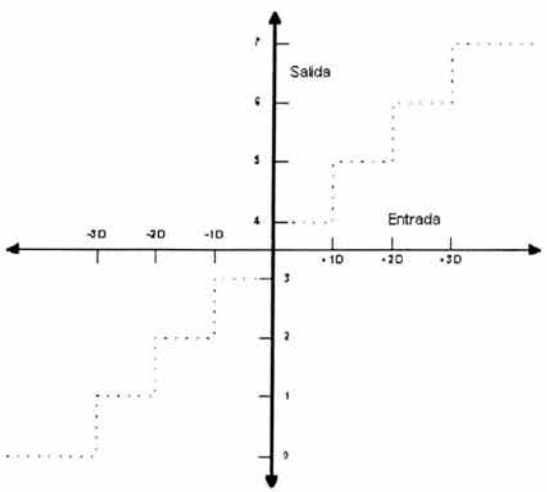


Figura 2.5 Gráfica de la relación entrada-salida de un cuantizador con decisión suave

Cabe mencionar que la manera de representar los datos es en forma decimal, y la forma de operar los datos al realizar la decodificación en forma binaria.

```
/* Esto inicializa un cuantizador que se adapta a la Eb/No */
void cuant_adap_ini(float es_ovr_n0) {

    int i, d;
    float es, sn_ratio, sigma;

    es = 1;
    sn_ratio = (float) pow(10.0, ( es_ovr_n0 / 10.0 ) );
    sigma =  (float) sqrt( es / ( 2.0 * sn_ratio ) );

    d = (int) ( 32 * 0.5 * sigma );

    for (i = -128; i < ( -3 * d ); i++)
        tabla_cuantizacion[i + 128] = 7;

    for (i = ( -3 * d ); i < ( -2 * d ); i++)
        tabla_cuantizacion[i + 128] = 6;

    for (i = ( -2 * d ); i < ( -1 * d ); i++)
        tabla_cuantizacion[i + 128] = 5;

    for (i = ( -1 * d ); i < 0; i++)
        tabla_cuantizacion[i + 128] = 4;

    for (i = 0; i < ( 1 * d ); i++)
        tabla_cuantizacion[i + 128] = 3;

    for (i = ( 1 * d ); i < ( 2 * d ); i++)
        tabla_cuantizacion[i + 128] = 2;

    for (i = ( 2 * d ); i < ( 3 * d ); i++)
        tabla_cuantizacion[i + 128] = 1;

    for (i = ( 3 * d ); i < 128; i++)
        tabla_cuantizacion[i + 128] = 0;
}
```

La idea de formar un arreglo con los valores de cuantización, es que *el índice del arreglo será el valor del símbolo de canal a cuantizar más un offset debido a que el índice debe ser un valor positivo*. Esto es, si el símbolo de canal toma un valor de -6, por ejemplo, entonces su valor cuantizado será obtenido del arreglo, y será el valor del elemento tabla_cuantizacion[-6+128], si el símbolo de canal es 10 entonces será el elemento tabla_cuantizacion[-10+128] y así para cada caso. Pero antes, este valor del símbolo de canal debe ser transformado, la transformación es debido a que este cuantizador asume que la media del valor del símbolo de canal es +/- 1, y traslada esto a un entero cuyo valor medio es +/- 32, con el fin de tener una mejor resolución en el rango de valores que toma el símbolo de canal a cuantizar y que va a ser comparado con el índice del arreglo *tabla_cuantizacion*. En total son 256 valores que toma el índice del arreglo, todos positivos, y dado que el valor del símbolo puede tomar valores negativos, entonces debemos sumarles un valor positivo (valor offset) para pasar todos posibles del símbolo de canal hacia los positivos.

La siguiente función realiza la transformación del símbolo de canal y obtiene valor cuantizado de la tabla de cuantización.

```
int cuant_suave(float simbolo_canal) {
    int x;
    x = (int) ( 32.0 * simbolo_canal );
    if (x < -128) x = -128;
    if (x > 127) x = 127;
    return(tabla_cuantizacion[x + 128]);
}
```

Veamos el siguiente ejemplo. Suponga que la relación señal a ruido es la siguiente:

$$es\_ovr\_n0 = 1dB$$

y se desea cuantizar el siguiente símbolo de canal

$$simbolo\_canal = -1.7$$

entonces la rutina *cuant_adap_ini* que construye la tabla de cuantización realiza los siguientes cálculos

$$sn\_ratio = 10.0 (es\_ovr\_n0 / 10.0) = 10 (1 / 10) = 1.26$$
$$sigma = sqrt(es / (2.0 * sn\_ratio)) = 0.63$$

el cálculo de "d" que nos sirve para el cálculo de los intervalos de decisión, implica aplicar un molde entero, para nuestro ejemplo queda lo siguiente

$$d = (int) (32 * 0.5 * sigma) = (int) (10.08) = 10$$

Por lo tanto la tabla de cuantización tendría una apariencia como la siguiente:

```
tabla_cuantizacion[0] =  7;
tabla_cuantizacion[1] =  7;
tabla_cuantizacion[2] =  7;
                .
                .
                .
tabla_cuantizacion[97] =  7;
tabla_cuantizacion[98] =  6;
tabla_cuantizacion[99] =  6;
tabla_cuantizacion[100] = 6;
                .
                .
                .
tabla_cuantizacion[107] = 6;
tabla_cuantizacion[108] = 5;
tabla_cuantizacion[109] = 5;
tabla_cuantizacion[110] = 5;
                .
                .
                .
tabla_cuantizacion[117] = 5;
tabla_cuantizacion[118] = 4;
tabla_cuantizacion[119] = 4;
tabla_cuantizacion[120] = 4;
                .
                .
                .
tabla_cuantizacion[127] = 4;
tabla_cuantizacion[128] = 3;
tabla_cuantizacion[129] = 3;
tabla_cuantizacion[130] = 3;
                .
                .
                .
tabla_cuantizacion[137] = 3;
tabla_cuantizacion[138] = 2;
tabla_cuantizacion[139] = 2;
tabla_cuantizacion[140] = 2;
```

```
.
.
tabla_cuantizacion[147] = 2;
tabla_cuantizacion[148] = 1;
tabla_cuantizacion[149] = 1;
tabla_cuantizacion[150] = 1;
.
.
tabla_cuantizacion[157] = 1;
tabla_cuantizacion[158] = 0;
tabla_cuantizacion[159] = 0;
.
.
tabla_cuantizacion[255] = 0;
```

luego, la rutina *cuant_suave* hará las siguientes operaciones finales y elige el valor cuantizado del símbolo de canal:

x = (int) (32.0 * simbolo_canal) = (int) (32.0 * (-1.7)) = -54
valor_retornado = tabla_cuantizacion[x + 128]

donde x = -54, entonces:

valor_retornado = tabla_cuantizacion[-54 + 128]
valor_retornado = tabla_cuantizacion[74] = 7

Finalmente, el valor cuantizado para un símbolo de canal recibido con valor –1.7 y una relación señal a ruido igual a 1 dB nos daría 7 (111_2). Cabe recordar que la tabla de cuantización se construye solo una vez cuando se invoca al proceso de decodificación de un mensaje y no para cada símbolo del mensaje.

2.2.5.2 Decodificación

Quizás la herramienta más importante para ayudar a entender el algoritmo de Viterbi es el diagrama de enrejado. La figura de abajo demuestra el diagrama de enrejado para nuestro codificador convolucional del ejemplo con tasa 1/2 K = 3, para un mensaje 15 bits

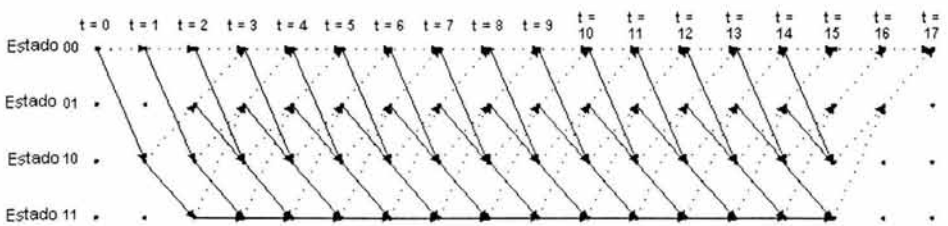


Figura 2.6 Diagrama de trellis del codificador convolucional del ejemplo con tasa 1/2 K = 3, para un mensaje 15 bits

Los cuatro estados posibles del codificador se representan como cuatro filas de puntos horizontales. Hay una columna de cuatro puntos para el estado inicial del codificador y uno para cada instante de tiempo durante el mensaje. Para un mensaje de 15 bits con dos bits que limpian la memoria del codificador (bits de limpieza), hay 17 instantes de tiempo además de t = 0, que representa la condición inicial del codificador. Las líneas continuas que conectan puntos en el diagrama, representan transiciones de estado cuando el bit de entrada es uno. Las líneas punteadas representan transiciones de estado cuando el bit de la entrada es un cero. Note la correspondencia entre las flechas en el diagrama de enrejado y la tabla de la transición de estados discutida anteriormente. También note que debido a que la condición inicial del codificador es el estado 00_2, y los dos bits que limpian la memoria son ceros, las flechas comienzan en el estado 00_2 y termine en el mismo estado 00_2.

Los bits de entrada del codificador y los símbolos de salida se muestran abajo del diagrama. Note la correspondencia entre los símbolos de salida del codificador y la tabla de salida discutidos anteriormente. Vamos a mirar esto más detalladamente, usando la versión ampliada de la transición entre un instante de tiempo al otro, mostrada abajo:

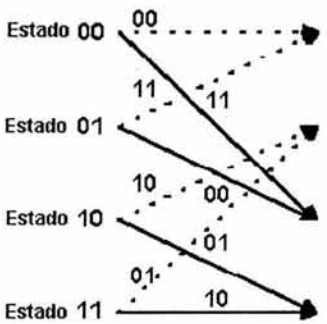


Figura 2.7 Detalle de la transición para un cierto instante de tiempo del diagrama de trellis

Los dos bits que etiquetan las líneas corresponden a los símbolos de canal de salida del codificador convolucional. Recuerde que las líneas punteadas representan casos donde la entrada del codificador es un cero, y las líneas sólidas representan casos donde la entrada del codificador es uno.

2.2.5.3 Implementación del decodificador

Para implementar el decodificador en software, el primer paso es construir las estructuras de datos alrededor de las cuales el algoritmo de decodificación va a ser implementado. Estas estructuras de datos son mejor implementadas como *arreglos*. Los seis arreglos primarios que necesitamos para el decodificador de Viterbi son como sigue:

- La tabla *estado siguiente* del codificador convolucional, la tabla de transición de estados del codificador. Las dimensiones de esta tabla (renglones x columnas) son $2^{(K-1)} \times 2^{k}$. Este arreglo necesita ser inicializado antes de comenzar con el proceso de decodificación.
- La tabla de salida del codificador convolucional. Las dimensiones de esta tabla (renglones x columnas) son $2^{(K-1)} \times 2^{k}$. Estos arreglos necesitan ser inicializados antes de comenzar con el proceso de decodificación.
- Un arreglo (tabla de entrada) mostrando para cada estado actual y estado siguiente del codificador convolucional, cuál valor de entrada (0 ó 1) debe producir el siguiente estado, dado el estado actual. Llamaremos a este arreglo la *tabla de entrada*. Las dimensiones de esta tabla (renglones x columnas) son $2^{(K-1)} \times 2^{(K-1)}$. Este arreglo necesita ser inicializado antes de comenzar con el proceso de decodificación.
- Un arreglo para almacenar la historia del estado predecesor para cada estado del codificador y almacenar hasta K x 5 + 1 pares de símbolos de canal recibidos. Llamaremos a esta tabla, la *tabla de historia de estados*. Las dimensiones de este arreglo (renglones x columnas) son $2^{(K-1)}$ x (K x 5 + 1) . Este arreglo no necesita ser inicializado antes de comenzar con el proceso de decodificación.
- Una tabla para almacenar las métricas del error acumulado para cada estado computado usando la operación sumar-comparar-seleccionar. Esta tabla es llamada *tabla de la métrica de error acumulada*. Las dimensiones de este arreglo (renglones x columnas) son $2^{(K-1)}$ x 2. Este arreglo no necesita ser inicializado antes de comenzar con el proceso de decodificación.
- Un arreglo para almacenar una lista de los estados determinados durante el trazado de camino. Este es llamado el *arreglo de la secuencia de estado*. Las dimensiones de este arreglo (renglones x columnas) son (K x 5) + 1. Este arreglo no necesita ser inicializado antes de comenzar con el proceso de decodificación.

Las tablas siguientes, son las tablas del *estado siguiente*, la *tabla de salida*, y la *tabla de entrada* respectivamente para el ejemplo de codificador convolucional de tasa 1/2 K = 3 :

	Estado siguiente, si	
Estado Actual	**Entrada = 0:**	**Entrada = 1:**
00	00	10
01	00	10
10	01	11

11	01	11

(a)

	Símbolos de la salida, si	
Estado Actual	**Entrada = 0:**	**Entrada = 1:**
00	00	11
01	11	00
10	10	01
11	01	10

(b)

	La entrada fue, dado el estado siguiente =			
Estado Actual	**00_2 = 0**	**01_2 = 1**	**10_2 = 2**	**11_2 = 3**
00_2 = 0	0	x	1	x
01_2 = 1	0	x	1	x
10_2 = 2	x	0	X	1
11_2 = 3	x	0	x	1

(c)

Tablas 2.4 Tablas de (a) estado siguiente, (b) tabla de salida, y (c)tabla de entrada, respectivamente, note que sus valores son escritas en forma binaria.

Se debe ahora ver, que con estas tres tablas se puede describir totalmente el comportamiento del codificador convolucional del ejemplo de tasa ½, K = 3, y que serán instrumento básico en la implementación del decodificador. La parte de código encargada de la construcción de estas estructuras se encuentra en la parte inicial del módulo decodificador de Viterbi. Primero se inician con valores cero todos los arreglos y después se les asignan sus valores correspondientes con las siguientes líneas:

```
for (j = 0; j < numero_estados; j++) {
    for (l = 0; l < n; l++) {
        sgte_edo = edo_sgte(j, l, memory_contents);
        input[j][sgte_edo] = l;

        /* Ahora computar la salida del codificador convolucional dado
           el número del estado actual y el valor de entrada */
```

```
        salida_rama[0] = 0;
        salida_rama[1] = 0;

        for (i = 0; i < K; i++) {
            salida_rama[0] ^= memory_contents[i] & g[0][i];
            salida_rama[1] ^= memory_contents[i] & g[1][i];
        }
        /* siguiente estado, dado el estado actual y la entrada  */
        sgte_edo[j][l] = sgte_edo;
        /* salida en decimal, dado el estado actual y la entrada */
        output[j][l] = bin2deci(salida_rama, 2);
    } /* fin del bucle for l */

} /* fin del bucle for j */
```

Se observa que se trata de dos ciclos, uno de los cuales se encuentra dentro de otro. El ciclo más externo (el que tiene j como índice) representa el *número de estado*, de tal manera que cuando j tenga el valor 1, por ejemplo, estaremos hablando del estado 1, en valor decimal (01 en binario) de hecho las matrices antes mencionadas serán utilizadas con valores decimales y no con valores binarios como fueron escritas anteriormente, de tal forma que las mismas matrices serán escritas de la siguiente manera;

	Estado siguiente, si	
Estado Actual	**Entrada = 0:**	**Entrada = 1:**
0	0	2
1	0	2
2	1	3
3	1	3

(a)

	Símbolos de la salida, si	
Estado Actual	**Entrada = 0:**	**Entrada = 1:**
0	0	3
1	3	0
2	2	1
3	1	2

(b)

	La entrada fue, dado el estado siguiente =			
Estado Actual	**$00_2 = 0$**	**$01_2 = 1$**	**$10_2 = 2$**	**$11_2 = 3$**
$00_2 = 0$	0	0	1	0
$01_2 = 1$	0	0	1	0
$10_2 = 2$	0	0	0	1
$11_2 = 3$	0	0	0	1

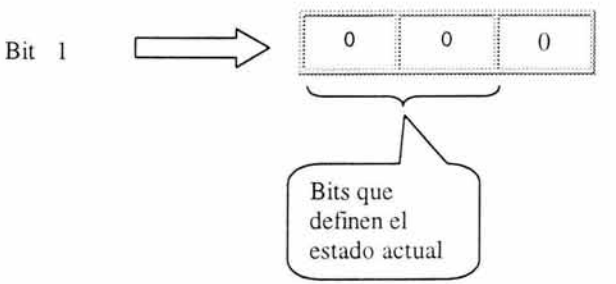
(c)

Tablas 2.5 Tablas de (a) estado siguiente, (b) tabla de salida, y (c)tabla de entrada, respectivamente, escritas con valores decimales

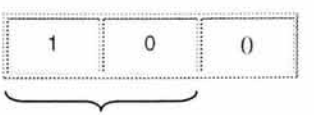
Para el ciclo interno, observamos que su índice es i, este índice representa el *bit de entrada* uno o cero. Lo primero que se realiza al entrar en este ciclo es encontrar el estado siguiente, esto se logra sabiendo que tenemos el estado actual y el bit de entrada y suponiendo que se inicia del estado 0 (00_2).

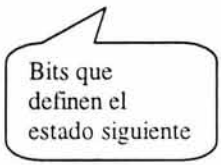
Veamos el siguiente ejemplo de cálculo de las tablas del decodificador, y empecemos con el cálculo del estado siguiente:

Para el inicio del ciclo, dado que la decodificación está ligada al proceso de codificación, utilizamos el mismo concepto de registro de desplazamiento. Entonces tenemos el registro de desplazamiento implementado con un arreglo, con puros elementos cero, como sigue:



y son justamente los dos primeros bits, los que definen el estado actual, para este caso tenemos el valor de j = 0 (estado = 0, dado que j representa el estado), y el valor del bit que entra con el valor de i=1 (dado que i representa el bit que entra). Para la ilustración anterior se eligió un bit que entra igual a "1" para ver el proceso con mayor claridad. Entonces el *estado siguiente* será:





Por lo que se observa que para el valor de "Estado actual" 0 (00_2) y un valor de entrada "1" el valor del "Estado siguiente" es 2 (10_2). Por lo que el programa de codificación, lo que realiza es el llamado a una rutina *edo_sgte*, que hace justamente el proceso que acabamos de ilustrar, el corrimiento de un bit en el registro de desplazamiento y las debidas conversiones de decimal a binario y viceversa, dado que los valores de los estados se manejan en forma decimal y su proceso es en forma binaria. El valor que retorna esta rutina se almacena en una variable sgte_edo, es decir el valor del estado siguiente.

La primera matriz que se actualiza, es la *matriz de entrada*, observamos que las dimensiones de esta matriz corresponden al número de estado actual, para el número de renglón, y al número de estado siguiente para el número columna es decir:

Input[estado_actual][estado_siguiente]

Para nuestro ejemplo la asignación quedaría:

Input[0][estado_siguiente=2] = valor de entrada =1

Posterior a actualizar a la matriz de entrada, se procede a calcular la salida de rama o lo que es lo mismo, los símbolos de salida del codificador dado el estado actual y el bit de entrada. Para este cálculo se utiliza un proceso muy similar al usado en el módulo de codificación, se aplican las operación lógicas AND Y XOR bit a bit mediante un ciclo FOR como fue descrito en el módulo de codificación convolucional, con la única diferencia de que no utilizamos un arreglo que funcione como registro circular sino que ahora tomamos los valores de un arreglo llamado *memory_contents* que es un arreglo que es actualizado cuando se calcula el estado siguiente con la rutina *edo_sgte*, y que contiene los datos del registro de desplazamiento para el estado siguiente. Los valores de salida calculados se almacenan en un arreglo temporal llamado *salida_rama*.

La siguiente tabla que se actualiza es la tabla *sgte_edo*, observamos que las dimensiones de esta matriz corresponden al número de estado actual, para el número de renglón, y al bit de entrada para el número columna es decir:

sgte_edo [estado_actual][bit de entrada]

Para nuestro ejemplo la asignación quedaría:

sgte_edo[0][1] = siguiente estado = 2

Por último, la siguiente tabla que se actualiza es la tabla *output*, para este caso utilizamos la salida del codificador guardada en el arreglo *salida_rama*, para el estado actual y bit de entrada correspondiente, pero además se aplica una función que convierte dicho valor de salida de su forma binaria a forma decimal. Observamos que al igual que la

matriz *sgte_edo,* las dimensiones de esta matriz corresponden al número de estado actual, para el número de renglón, y al bit de entrada para el número columna es decir:

output [estado_actual][bit de entrada]

Para nuestro ejemplo la asignación quedaría:

output [0][1] = salida_rama = 3 = 11_2

Una vez establecidas las estructuras de datos alrededor de las cuales el algoritmo de decodificación va a ser implementado, el siguiente paso es cargar los datos que se reciben como parámetro de entrada y hacer su cuantización. Estos datos que se reciben como símbolos de canal son producidos por el codificador convolucional pero con la adición de ruido, por lo que los valores de símbolo no son enteros sino de punto flotante. La idea del cuantizador es precisamente convertir estos valores de punto flotante a valores enteros (discretos). Para lograr esto se invoca a la rutina *cuant_suave*, que convierte el valor de punto flotante que recibe como parámetro o su valor entero correspondiente, mediante una comparación de rangos establecidos por la relación señal a ruido que es también recibida como parámetro y que varía durante el transcurso de la prueba, tal como se indico en la sección anterior.

```
matriz_salida_canal = malloc( long_canal * sizeof(int) );
    if (matriz_salida_canal == NULL) {
        printf("\nsdvd.c: No puedo asignar suficiente espacio de memoria para
          matriz_salida_canal!  Abortando.... ");
        exit(1);
    }
```

con el propósito de una fácil operación de los índices del arreglo que almacena los datos, vamos a modificar el orden de los símbolos de canal recibidos de tal manera que los símbolos correspondientes a un bit codificado esten separados una cantidad de *long_canal* lugares en el arreglo, esto facilitara la manipulación de los datos como veremos más adelante

```
    long_canal = long_canal / n;
    /* cuantizar la salida del canal (convertir de real a  entero corto) */
    for (t = 0; t < (long_canal * n); t += n) {
        for (i = 0; i < n; i++)
            *(matriz_salida_canal + (t / n) + (i * long_canal) ) =
                cuant_suave( *(vector_salida_canal + (t + i) ) );
    } /* fin del bucle-for t */
```

Ahora vamos a comenzar a mirar cómo trabaja realmente el algoritmo decodificador de Viterbi. El proceso de decodificar comienza con la construcción de la métrica del error acumulado (matriz metric_err_acum[TWOTOTHEM][2]) para un cierto número de pares recibidos de símbolos de canal, y la historia de qué estados precedieron (matriz historia_edo[TWOTOTHEM][K * 5 + 1]) a los estados en cada instante de tiempo t con la más pequeña métrica del error acumulado.

El código fuente utiliza entradas de decisión suave para alcanzar un funcionamiento mejor. Cada vez que recibimos un par de símbolos del canal, vamos a computar una métrica para medir la "distancia" entre lo que recibimos y todos los pares posibles de símbolos de canal que se pudieron recibir. Los valores de la métrica que computamos en cada instante de tiempo para las trayectorias entre los estados en el tiempo anterior inmediato y los estados en el instante de tiempo actual se llaman métrica de rama. Para el primer instante de tiempo, vamos a guardar estos resultados como valores de la "medida acumulada del error", asociados a los estados. Para el segundo instante de tiempo, la métrica acumulada del error será computada agregando la métrica acumulada del error anterior a la métrica actual de rama, y así sucesivamente. Revisemos paso a paso el código que implementa este proceso.

Primero, el proceso está dentro de un ciclo, cada iteración representa un instante de tiempo y se procesan n símbolos de canal que corresponden a un bit codificado, para este ejemplo supongamos n = 2, para el codificador convolucional de tasa ½, K = 3

```
for (t = 0; t < long_canal - m; t++) {
                .
                .
                .
        Proceso de decodificación
                .
                .
} /* fin del bucle-for 't' */
```

Yendo de t = 0 a t = 1, si observamos el enrejado, hay solamente dos pares posibles de símbolos de canal que se pudieron recibir: 00_2 y 11_2 . Esto es porque sabemos que el codificador convolucional fue inicializado con el estado todos ceros, y que dado un bit de entrada igual a uno o cero, hay solamente dos estados hacia los que ocurriría la transición y dos salidas posibles del codificador. Estas salidas posibles del codificador son 00_2 y 11_2 . La variable *step* llevará el control de cuales son los estados para los cuales la métrica acumulada del error será computada, donde "j" del segundo ciclo representa *el estado inicial* de una transición que va a ser computada para el instante de tiempo "t". Esto explica la siguiente parte del código en el inicio del decodificador

```
for (t = 0; t < long_canal - m; t++) {

    if (t <= m)
        /* asumimos comienzo con ceros, de tal manera que solo computamos
           caminos desde el estado todos-ceros */
        step = pow(2, m - t * 1);
     else
        step = 1;
     /* vamos a usar el arreglo de historia de estados como un bufer circular
        de tal manera que no desplacemos todo hacia la izquierda después de
        que cada bit sea procesado, esto significa que necesitamos un
         apuntador apropiado */
     /* ajustar el apuntado del arreglo de historia de estados para este tiempo t*/
     sh_ptr = (int) ( ( t + 1 ) % (profun_trellis + 1) );
```

```
        /* repetir para cada estado posible */
        for (j = 0; j < numero_estados; j+= step) {

            Proceso de decodificación
                    .
                    .

        } /* fin del bucle-for 'j' */

        Proceso de decodificación
                    .
                    .

    } /* fin del bucle-for 't' */
```

Posteriormente se entra a un nuevo ciclo, este ciclo es el que realiza propiamente el cálculo de la métrica de error de la n-upla de los símbolos de canal (el par de símbolos de canal para nuestro ejemplo, dado que n = 2) y la salida de rama de la transición de estado en turno. Tenemos que el índice "l" de este ciclo, indica para qué transición de estado se va a realizar el cálculo de la métrica, el estado inicial lo indica el valor de "j" y el valor final esta implícito en el valor de "l", si l = 1 indica que se tomará la salida de rama o salida del codificador que resulta cuando el bit de entra es un 1, por el contrario si l = 0, se realizará el proceso tomando la salida de rama o salida del codificador cuando el bit de entrada es un 0. El valor de la salida de rama o salida del codificador es obtenido de la matriz *output*, y el valor del estado final o estado siguiente es obtenido con la matriz *sgte_edo*, ambas matrices fueron descritas con anterioridad, y utilizan los valores de "j" (estado actual) y de "l" (bit de entrada) como indices . La implementación en código queda:

```
for (t = 0; t < long_canal - m; t++) {

    if (t <= m)
        /* asumimos comienzo con ceros, de tal manera que solo computamos
           caminos desde el estado todos-ceros */
        step = pow(2, m - t * 1);
    else
        step = 1;
    /* vamos a usar el arreglo de historia de estados como un bufer circular
       de tal manera que no desplacemos todo hacia la  izquierda después de
       que cada bit sea procesado,esto significa que necesitamos un
        apuntador apropiado */
    /* ajustar el apuntado del arreglo de historia de estados para este tiempo t*/
    sh_ptr = (int) ( ( t + 1 ) % (profun_trellis + 1) );

    /* repetir para cada estado posible */
    for (j = 0; j < numero_estados; j+= step) {
        /* repetir para cada n-upla de salida del codificador convolucional posible */
        for (l = 0; l < n; l++) {
            metrica_rama = 0;

            /* computa métrica de rama por símbolo de canal, y suma
               para todos los símbolos de canal en la n-upla de salida del
               codificador convolucional */

            #ifdef FASTACS

            /* convierte la representación decimal de la salida del a binaria */
            salida_binaria[0] = ( output[j][l] & 0x00000002 ) >> 1;
            salida_binaria[1] = output[j][l] & 0x00000001;
```

```
            /* computa la métrica de rama por símbolo de canal, y suma
               para todos los símbolos de canal en la n-upla de salida del
               codificador convolucional */
            metrica_rama = metrica_rama + abs( *( matriz_salida_canal +
                ( 0 * long_canal + t ) ) - 7 * salida_binaria[0] ) +
                                          abs( *( matriz_salida_canal +
                ( 1 * long_canal + t ) ) - 7 * salida_binaria[1] );
            #endif
```

Proceso de decodificación

```
                .
                .

        } /* fin del bucle-for '1' */
    } /* fin del bucle-for 'j' */
```

Proceso de decodificación

```
            .
            .

} /* fin del bucle-for 't' */
```

El siguiente proceso es ahora escoger el camino sobreviviente, el que tenga la métrica de error acumulada más pequeña, tenemos entonces, la matriz *metric_err_acum* en donde se almacena la métrica de error acumulada, que resulta entre la salida de rama de la transición de estados y la n-upla de símbolos de canal que se están analizando. El número de renglón de esta matriz corresponde al número de estado final de la transición de estado, por ejemplo el elemento *metric_err_acum*[2][x] corresponde a la métrica de error acumulada de una cierta transición de estados cuyo estado final es el estado cero. En el programa, el estado final, para un estado inicial " j " y un bit de entrada "1", es calculado con la matriz *sgte_edo* [j][1]. Además, durante el computo realizado para escoger el camino sobreviviente para un cierto tiempo t, la columna 1 contiene temporalmente el valor de la métrica acumulada más pequeña, es decir el camino sobreviviente, y la columna 0 contiene el camino sobreviviente que resulta del análisis de otros estados, esto es, si observamos el enrejado para un cierto estado, tenemos al menos dos posibles estados de los cuales podemos provenir, y dado que el cómputo del camino sobreviviente se realiza de un estado inicial a la vez debemos tener un arreglo de memoria para almacenar los resultados del análisis de otros estados iniciales, este arreglo es la columna 0 de la matriz *metric_err_acum* y la columna 1 guarda los resultados para el estado que se está analizando, es por eso que se hace una comparación entre elementos de la columna 1 y la columna 0. Observe que si la métrica de error acumulada para el estado que se esta analizando es menor que la de otros estado en el instante de tiempo que se esta analizando esta métrica se guarda en la columna 1 de la matriz *metric_err_acum.* y se guarda el número de estado que se está analizando en la matriz *historia_edo*, de lo contrario no se realiza ninguna operación de guardado y se sigue con otra iteración de análisis de métrica de error acumulada. Veamos el la sección de código que implementa este procedimiento

```
            /* ahora escogemos el camino sobreviviente ( el que tenga la
               métrica de error acumulada más pequeña) */
            if ( metric_err_acum[ sgte_edo[j][1] ] [1] > metric_err_acum[j][0] +
                metrica_rama ) {
```

```
                /* guarda un valor de métrica acumulada para el estado sobreviviente */
                metric_err_acum[ sgte_edo[j][1] ] [1] = metric_err_acum[j][0] +
                    metrica_rama;

                /* actualiza el arreglo de historia de estado con el número de
                   estado del sobreviviente */
                c[ sgte_edo[j][1] ] [sh_ptr] = j;

            } /* fin de la sentencia if */
```

luego tenemos la parte de código donde la información temporal de la columna 1 se pasa a la columna 0 para su posterior comparación con los resultados del cómputo de otros estados, además , en los elementos de la columna 1 se guarda el valor MAXINT, que es el valor más grande que un dato tipo entero puede soportar, esto se hace con la finalidad de tener una especie de bandera para indicar que en esa localidad no hay algún valor resultado del computo de algún estado y que al realizar posteriores comparaciones para elegir la métrica de error acumulada podamos tener una mala decisión , dado que siempre se elige la cantidad menor.

```
        /* para todas las filas de la matriz metric_err_acum, mover columna 2 a la
           columna 1 y poner bandera en columna 2 */
        for (j = 0; j < numero_estados; j++) {
            metric_err_acum[j][0] = metric_err_acum[j][1];
            metric_err_acum[j][1] = MAXINT;
        } /* fin del bucle-for 'j' */
```

y ahora veamos los segmentos de código incrustado en la rutina que estamos analizando

```
for (t = 0; t < long_canal - m; t++) {

    if (t <= m)
        /* asumimos comienzo con ceros, de tal manera que solo computamos
           caminos desde el estado todos-ceros */
        step = pow(2, m - t * 1);
    else
        step = 1;
    /* vamos a usar el arreglo de historia de estados como un bufer circular
       de tal manera que no desplacemos todo hacia la  izquierda después de
       que cada bit sea procesado,esto significa que necesitamos un
       apuntador apropiado */
    /* ajustar el apuntado del arreglo de historia de estados para este tiempo t*/
    sh_ptr = (int) ( ( t + 1 ) % (profun_trellis + 1) );

    /* repetir para cada estado posible */
    for (j = 0; j < numero_estados; j+= step) {
        /* repetir para cada n-upla de salida del codificador convolucional posible */
        for (l = 0; l < n; l++) {
            metrica_rama = 0;

            /* computa métrica de rama por símbolo de canal, y suma
               para todos los símbolos de canal en la n-upla de salida del
               codificador convolucional */

            #ifdef FASTACS

            /* convierte la representación decimal de la salida del a binaria */
            salida_binaria[0] = ( output[j][l] & 0x00000002 ) >> 1;
            salida_binaria[1] = output[j][l] & 0x00000001;

            /* computa la métrica de rama por símbolo de canal, y suma
               para todos los símbolos de canal en la n-upla de salida del
               codificador convolucional */
            metrica_rama = metrica_rama + abs( *( matriz_salida_canal +
```

```
                ( 0 * long_canal + t ) ) - 7 * salida_binaria[0] ) +
                                        abs( *( matriz_salida_canal +
                ( 1 * long_canal + t ) ) - 7 * salida_binaria[1] );
            #endif

            /* ahora escogemos el camino sobreviviente ( el que tenga la
               métrica de error acumulada más pequeña) */
            if ( metric_err_acum[ sgte_edo[j][1] ] [1] > metric_err_acum[j][0] +
                metrica_rama ) {

                /* guarda un valor de métrica acumulada para el estado sobreviviente */
                metric_err_acum[ sgte_edo[j][1] ] [1] = metric_err_acum[j][0] +
                    metrica_rama;

                /* actualiza el arreglo de historia de estado con el número de
                   estado del sobreviviente */
                historia_edo[ sgte_edo[j][1] ] [sh_ptr] = j;

            } /* fin de la sentencia if */
        ) /* fin del bucle-for 'l' */
    } /* fin del bucle-for 'j' */

    /* para todas las filas de la matriz metric_err_acum, mover columna 2 a la
       columna 1 y poner bandera en columna 2 */
    for (j = 0; j < numero_estados; j++) {
        metric_err_acum[j][0] = metric_err_acum[j][1];
        metric_err_acum[j][1] = MAXINT;
    } /* fin del bucle-for 'j' */
```

Proceso de decodificación

```
                .
                .
} /* fin del bucle-for 't' */
```

Una vez que se acumule la información, el decodificador de Viterbi está listo para reconstruir la secuencia de los bits que entraron al codificador convolucional cuando el mensaje fue codificado para la transmisión. Cabe mencionar, que antes de terminar de procesar todos los símbolos de canal que se recibieron en el arreglo de entrada, puede suceder que la tabla historia_edo(tabla donde se guarda la historia de estados) se llene, dado que tiene una longitud igual a la longitud del enrejado, por lo que será necesario realizar una reconstrucción de secuencia de datos previa, en el que solo se decodifica el símbolo de canal del mensaje más antiguo que entro a la decodificación y se desplaza toda la información de la secuencia de datos un lugar, es decir se pierde la información del bit que fue codificado con la finalidad de seguir con la decodificación de los símbolos de canal que falten de procesar. Este proceso de decodificación previo tiene una estructura de codificación muy similar a la decodificación final, y esto es logrado por los pasos siguientes:

- Primero, seleccionar el estado que tiene la métrica de error acumulado más pequeña y guarde el número de estado de ese estado.
- Realice iterativamente el paso siguiente hasta que el principio del enrejado se alcance: Trabajando al revés a través de la tabla de la historia del estado, para el estado seleccionado, seleccione un nuevo estado el cual esté listado en la tabla de la

historia de estados como el precursor a ese estado. Guarde el número del estado de cada estado seleccionado. Este paso se llama trazo de camino[2] de retorno.

- Ahora trabaje adelante a través de la lista de los estados seleccionados guardados en los pasos anteriores. Observe qué bit de entrada corresponde a una transición de cada estado precursor a su estado sucesor. Éste es el bit que debió ser codificado por el codificador convolucional.

Veamos el código completo de este proceso de decodificación

```
for (t = 0; t < long_canal - m; t++) {

    if (t <= m)
        /* asumimos comienzo con ceros, de tal manera que solo computamos
           caminos desde el estado todos-ceros */
        step = pow(2, m - t * 1);
     else
        step = 1;
     /* vamos a usar el arreglo de historia de estados como un bufer circular
        de tal manera que no desplacemos todo hacia la  izquierda después de
        que cada bit sea procesado,esto significa que necesitamos un
         apuntador apropiado */
     /* ajustar el apuntado del arreglo de historia de estados para este tiempo t*/
     sh_ptr = (int) ( ( t + 1 ) % (profun_trellis + 1) );

     /* repetir para cada estado posible */
     for (j = 0; j < numero_estados; j+= step) {
         /* repetir para cada n-upla de salida del codificador convolucional posible */
         for (l = 0; l < n; l++) {
             metrica_rama = 0;

             /* computa métrica de rama por símbolo de canal, y suma
                para todos los símbolos de canal en la n-upla de salida del
                codificador convolucional */

             #ifdef FASTACS

             /* convierte la representación decimal de la salida del a binaria */
             salida_binaria[0] = ( output[j][l] & 0x00000002 ) >> 1;
             salida_binaria[1] = output[j][l] & 0x00000001;

             /* computa la métrica de rama por símbolo de canal, y suma
                para todos los símbolos de canal en la n-upla de salida del
                codificador convolucional */
             metrica_rama = metrica_rama + abs( *( matriz_salida_canal +
                 ( 0 * long_canal + t ) ) - 7 * salida_binaria[0] ) +
                                          abs( *( matriz_salida_canal +
                 ( 1 * long_canal + t ) ) - 7 * salida_binaria[1] );
             #endif

             /* ahora escogemos el camino sobreviviente ( el que tenga la
                métrica de error acumulada más pequeña) */
             if ( metric_err_acum[ sgte_edo[j][l] ] [l] > metric_err_acum[j][0] +
                 metrica_rama ) {

                 /* guarda un valor de métrica acumulada para el estado sobreviviente */
                 metric_err_acum[ sgte_edo[j][l] ] [l] = metric_err_acum[j][0] +
                     metrica_rama;

                 /* actualiza el arreglo de historia de estado con el número de
                    estado del sobreviviente */
                 historia_edo[ sgte_edo[j][l] ] [sh_ptr] = j;

             } /* fin de la sentencia if */
```

[2] A este proceso, en inglés, se le conoce como traceback

```
        } /* fin del bucle-for 'l' */
    } /* fin del bucle-for 'j' */

    /* para todas las filas de la matriz metric_err_acum, mover columna 2 a la
       columna 1 y poner bandera en columna 2 */
    for (j = 0; j < numero_estados; j++) {
        metric_err_acum[j][0] = metric_err_acum[j][1];
        metric_err_acum[j][1] = MAXINT;
    } /* fin del bucle-for 'j' */

    /* ahora comenzar el regreso, si tenemos lleno el enrejado */
    if (t >= profun_trellis - 1) {

        /* inicializa el vector de secuencia de estado */
        for (j = 0; j <= profun_trellis; j++)
            secuencia_edo[j] = 0;
        /* encuentra el elemento de la historia de estado con la mínima métrica
           de error acumulado */
        x = MAXINT;
        for (j = 0; j < ( numero_estados / 2 ); j++) {

            if ( metric_err_acum[j][0] < metric_err_acum[numero_estados - 1 - j][0] ) {
                xx = metric_err_acum[j][0];
                hh = j;
            }
            else {
                xx = metric_err_acum[numero_estados - 1 - j][0];
                hh = numero_estados - 1 - j;
            }
            if ( xx < x) {
                x = xx;
                h = hh;
            }

        } /* fin del bucle-for 'j' */

        /* ahora elegimos el punto de comienzo para el regreso */
        secuencia_edo[profun_trellis] = h;

        /* ahora trabajamos hacia atrás desde el final del enrejado a el
           estado más viejo en el enrejado para determinar el camino óptimo .
           El propósito de esto es determinar la secuencia de estados más probable
           en el codificador basados en qué símbolos de canal recibimos. */
        for (j = profun_trellis; j > 0; j--) {
            sh_col = j + ( sh_ptr - profun_trellis );
            if (sh_col < 0)
               sh_col = sh_col + profun_trellis + 1;

            secuencia_edo[j - 1] = historia_edo[ secuencia_edo[j] ] [sh_col];
        } /* fin del bucle-for 'j' */
        /* ahora proponemos que secuencia de entrada corresponde a la
           secuencia de estados en el camino óptimo */
        *(matriz_salida_decod + t - profun_trellis + 1) =
            input[ secuencia_edo[0] ] [ secuencia_edo[1] ];

    }/* fin de la sentencia if */

} /* fin del bucle-for 't' */
```

Retomando el ejemplo que se mencionó en la parte inicial del análisis del decodificador con la secuencia de datos de entrada: 0101110010100012. Tendríamos su secuencia de datos correspondiente de salida del codificador: $00\ 11\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 10\ 00\ 10\ 11\ 00\ 11\ 10\ 11_2$. Entonces, como producto intermedio del proceso de decodificación tendríamos la matriz de métrica acumulada (más los dos bits que limpian) para el mensaje completo en cada tiempo t:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Estado 00_2		0	2	3	3	3	3	4	1	3	4	3	3	2	2	4	5	2
Estado 01_2			3	1	2	2	3	1	4	4	1	4	2	3	4	4	2	
Estado 10_2		2	0	2	1	3	3	4	3	1	4	1	4	3	3	2		
Estado 11_2			3	1	2	1	1	3	4	4	3	4	2	3	4	4		

Tabla 2.6 Tabla de métrica acumulada para el mensaje del ejemplo

Es interesante observar que para este ejemplo de decodificador de Viterbi con entrada de decisión dura, la métrica de error acumulado más pequeño en el estado final indica cuántos errores de símbolos de canal ocurrieron.

Además, la siguiente tabla de la historia de estados muestra los estados precursores sobrevivientes para cada estado en cada tiempo t:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Estado 00_2	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0	0	1
Estado 01_2	0	0	2	2	3	3	2	3	3	2	2	3	2	3	2	2	2	0
Estado 10_2	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	0	0	0
Estado 11_2	0	0	2	2	3	2	3	2	3	2	2	3	2	3	2	2	0	0

Tabla 2.7 Tabla de la historia de estados precursores para el mensaje del ejemplo

La tabla siguiente muestra los estados seleccionados cuando se trazó la trayectoria a través de la tabla de estados sobrevivientes mostrada arriba:

t =	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	0	0	2	1	2	3	3	1	0	2	1	2	1	0	0	2	1	0

Tabla 2.8 Tabla de estados sobrevivientes final para el mensaje del ejemplo

Usando la tabla que mapea las transiciones de estado a las entradas que las causaron, podemos recrear el mensaje original. Así es como esta tabla se mira para nuestro ejemplo de codificador convolucional de tasa 1/2 K = 3 :

	La entrada fue, dado el estado siguiente =			
Estado Actual	**$00_2 = 0$**	**$01_2 = 1$**	**$10_2 = 2$**	**$11_2 = 3$**
$00_2 = 0$	0	0	1	0
$01_2 = 1$	0	0	1	0
$10_2 = 2$	0	0	0	1
$11_2 = 3$	0	0	0	1

Tabla 2.9 Tabla que relaciona las transiciones de estado a las entradas que las causaron.

De esta forma ahora tenemos todas las herramientas para reconstruir el mensaje original que nosotros recibimos:

t =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1

Tabla 2.10 Tabla que muestra el mensaje reconstruido

Los dos bits de *limpieza* son descartados para este último mensaje reconstruido.

Para el mensaje de ejemplo de 15 bits, se construyó el enrejado para el mensaje entero antes de iniciar el trazado de camino. Para mensajes más largos, o datos continuos, esto no es práctico o deseable, debido a las limitaciones de memoria y al retardo del decodificador. Las investigaciones han demostrado que una profundidad de trazado de K x 5 es suficiente para la decodificación de Viterbi con el tipo de códigos que se han mencionado. Cualquier profundidad de trazado mayor incrementa el retraso de la decodificación y los requerimientos de memoria del decodificador, mientras que no hay un mejoramiento significativo en el desempeño del decodificador.

Capítulo 3

Implantación del Sistema de Transmisión

Capítulo 3. Implantación del sistema de transmisión

Entre un usuario que envía caracteres en una terminal (una PC con un programa de terminal) y otro que los recibe casi inmediatamente en su pantalla hay mucho más que un simple puerto serie. Para la realización del sistema debemos tener en cuenta las siguientes consideraciones:

- El programa de terminal, que programa el controlador del puerto serie para recibir y enviar caracteres. Este programa terminal servirá también como controlador, que convierte los caracteres a enviar en cadenas de bits, las pone en la línea correspondiente y, en el sentido inverso, es capaz de interpretar la señal que le llega, convertirla en bits y juntarla en Bytes.
- Una interfaz de hardware inalámbrica, que permita realizar la transmisión y recepción del flujo de información serial, bajo las especificaciones RS-232 del puerto serie de forma transparente.

Los anteriores puntos, son las principales vertientes sobre las que gira la implementación del sistema de transmisión desarrollado y que a continuación se presenta

3.1 El programa terminal

En este caso se trata de mostrar la estructura de software necesaria cuando se quieren transferir ciertas cantidades de información (no tienen que ser necesariamente archivos) a través de una conexión en serie y, por ese motivo tiene que introducirse un protocolo que permita la localización y corrección de errores.

Antes de Analizar el código del programa terminal resulta útil revisar algunos conceptos para el mejor entendimiento del programa en general

3.1.1 Comunicación asíncrona

En la comunicación a través de líneas eléctricas cabe distinguir entre comunicación síncrona y asíncrona. Se habla de comunicación síncrona cuando el emisor y el receptor disponen de una señal de pulsos común que ayuda a coordinar todas sus acciones. Por ejemplo, que el emisor inmediatamente antes de un nuevo pulso envíe el siguiente bit a la línea y el receptor sepa que con la llegada del pulso puede recibirlo. En este caso, emisor y receptor están sincronizados.

Para la sincronización se precisa de una señal a través de la cual el emisor y el receptor intercambian la señal del pulso que sincroniza los momentos en que un cambio de señal debe interpretarse como un dato. Pero en la transmisión serie a través de un cable de dos líneas, no podemos disponer de un cable para la señal de sincronización, ya que ambas están ocupadas por los datos .

La sincronización por tanto, en una transmisión serie, debe llevarse a cabo a través de la línea de datos, con los datos mismos. Por este motivo se intercalan antes y después de

los datos, que pueden consistir en palabras de entre cinco y ocho bits, informaciones de estado según el estandar RS-232.

El número de estas informaciones lo determinan emisor y receptor al estructurar la conexión mediante la correspondiente programación de sus puertos serie, para lo cual deben estar de acuerdo, de lo contrario la comunicación no funcionaría.

A los bits de datos puede acompañarles lo que se denomina un bit de paridad con cuya ayuda se pueden descubrir errores en la transmisión. Para ello se diferencia entre paridad par e impar. En la paridad par la palabra de datos a transmitir se completa con el bit de paridad de manera que el número de bits enviados sea par. Si por ejemplo la palabra de datos que queremos enviar contiene tres bits con el valor 1, el bit de paridad será también 1, pues así el número de bits total aumenta con este bit adicional hasta cuatro y se consigue así una paridad par. Si por el contrario la palabra a transmitir tuviera un número par de bits con valor 1, el bit de paridad contendría el valor cero. Lo mismo pero al revés sucede con el bit de paridad en la paridad impar, de manera que el número de bits transmitidos con valor 1 sea impar.

El reconocimiento de la paridad, a pesar de que está bien pensado, algunas veces se ejecuta en balde, dado que los errores sólo se pueden detectar cuando por el camino se pierde un sólo bit de datos o un número impar de ellos, llegando no con su contenido que era el valor uno, sino con un cero (o al revés). Pero si en la transmisión son implicados un número par de bits de datos, la aritmética de la paridad seguirá siendo correcta, a pesar de haberse dado un error de transferencia.

Por este motivo, en la práctica de la comunicación de datos a través de correo informático, se evita la inserción automática de bits de paridad, lo cual resulta sencillo con la programación adecuada del controlador del puerto serie. En su lugar se emplean protocolos de más alto nivel que permiten descubrir errores en la transmisión de bloques de datos más grandes, por ejemplo efectuando sumas de comprobación.

Al contrario que el bit de paridad, el stop bit no es opcional pues indica la finalización de la transmisión de datos de una palabra de datos. El protocolo de transmisión de datos permite 1, 1.5 y 2 stop bit, dependiendo de la longitud de la palabra.

La posibilidad de trabajar con 1.5 stop bit puede parecer desconcertante en una primera impresión, sin embargo abre las puertas sobre una pregunta que durante mucho tiempo no se pudo contestar: ¿Cómo puede saber el receptor cuándo debe interpretar como un bit de datos el estado actual de la línea? Con lo que volvemos sobre el tema de la sincronización, pues cuando por ejemplo, el receptor intenta leer un bit en el mismo momento en que el emisor ya ha enviado el siguiente por la línea, no se puede predecir si se está leyendo el bit correcto.

La respuesta a todo esto la suministra la velocidad de transmisión medida en baudios, que debe estar configurada al mismo valor tanto para el emisor como para el receptor si se pretende que la transmisión funcione. Un baudio es un bit por segundo, por lo que las velocidades de transmisión habituales de hoy en día, como los 9600 baudios, no significan otra cosa que una velocidad de transmisión de 9600 bits por segundo. De esta manera se puede saber también que longitud tiene un bit, es decir 1/9600 segundos. Así, el receptor lee cada 1/9600 de segundo el estado actual de la línea de transmisión de datos y transforma su contenido dependiendo de si está en high o en low en un bit 1 o 0.

Pero sólo con esto no se consigue que el emisor y el receptor estén sincronizados pues todavía se puede dar el caso arriba mencionado en que el receptor intenta leer un bit

cuando el emisor ya ha enviado el siguiente por la línea. Para evitar esto aparece en escena el start bit o bit de inicio. Cuando el receptor detecta el start bit sabe que la transmisión ha comenzado y es a partir de entonces que debe leer las señales de la línea a distancias concretas de tiempo, en función una velocidad determinada. Para que no se pierda esta sincronización al cabo del tiempo debido a pequeñas diferencias entre el emisor y el receptor a la hora de contar las fracciones de tiempo, se envía un nuevo start bit con cada palabra de datos, de modo que emisor y receptor están continuamente sincronizándose.

Una vez sabido esto, ahora ya se puede contestar a la pregunta de cómo puede darse una longitud de stop bit de 1,5 bits: poniendo la línea en high o low al cabo de 1.5 veces el tiempo de transmisión de bits determinado. Con el stop bit concluye la transmisión de un carácter. La línea permanece a partir de ese momento en lo que se denomina marking condition es decir, en high, hasta que se tiene que transmitir un nuevo carácter para lo que debe transferirse un start bit lo cual a su vez precisa que la línea esté en low.

La transmisión de datos sólo funciona, sin embargo, cuando los diferentes parámetros variables del protocolo son conocidos tanto por el emisor como por el receptor. Lo primero es fijar el Baudrate, es decir, el número de bits por segundo. El margen permitido se encuentra entre 75 baudios y 115200 baudios, que es el máximo ratio de transmisión que soporta un puerto serie de una PC.

El número de bits de datos a transmitir depende de los datos a transmitir. Si se quieren transferir datos en ASCII, es suficiente con siete bits de datos, pues este conjunto de caracteres consta sólo de 128 códigos. Si por el contrario se quiere utilizar el extenso conjunto de caracteres de un PC, con sus 256 símbolos o llevar a cabo transmisiones binarias, deben utilizarse palabras de datos de ocho bits. Además puede definirse si se quiere o no llevar a cabo una comprobación de paridad y, si es el caso, si se trabajará con paridad par o impar. Ambos procedimientos ofrecen la misma (in)seguridad.

Por último debe definirse el número de stop bit. Si sólo se utiliza un stop bit, el siguiente carácter se transmitirá antes que con la utilización de dos stop bit. De todos modos esta selección se da muy raramente en la práctica.

3.1.2 Uso del BIOS

Mucho antes de que cualquier Sistema Operativo sea instalado en el disco duro, la PC ya dispone de un sistema operativo, que le ha sido "grabado": el BIOS. Está contenido en un circuito ROM, que se encuentra en la placa principal, y por ello está ya presente durante el primer arranque de un sistema.

Este sistema básico de entrada y salida (BIOS es acrónimo de «basic input output system») dispone de las funciones principales que necesita un programa para la comunicación con el hardware del PC y los dispositivos periféricos conectados. Allí se encuentran tanto funciones para la transferencia de caracteres en el puerto serie, como rutinas para el acceso a la pantalla o la gestión de la fecha y la hora.

Para la programación de la PC, los servicios del BIOS son tan esenciales que no sólo los programas de aplicación serían impensables sin ellos, sino el mismo WINDOWS no podría existir. Ya que el BIOS aísla el software de las características específicas del hardware y por ello le quita mucho trabajo a la hora de controlar los diferentes dispositivos. Pero no sólo es la comodidad que hace que el programador eche mano de las funciones del

BIOS. Sobre todo son las diferencias en el hardware de los diferentes fabricantes de PC que, a pesar de toda la «compatibilidad», existen naturalmente.

El BIOS pone a disposición una interfaz de comunicación estándar, para acceder a los diferentes componentes de hardware del sistema, de forma que si un sistema dispone de un disco duro de 40 GB, si es de IBM, Dell o Compaq; la interfaz de las funciones del disco duro del BIOS y la forma de trabajo es siempre la misma, y sólo de ello se trata. Que dentro del BIOS todo esto puede tener un aspecto diferente en cada caso, no le ha de interesar al programador.

El estándar que IBM definió en asuntos del BIOS, define la forma en que se llaman las diferentes funciones del BIOS (llámense rutinas), y como se realiza el paso de parámetros. Para aislar al invocador de la situación de las funciones del BIOS, se eligió el concepto de interrupciones como mecanismo de llamada. Ya que no se puede disponer de una interrupción para cada función, (sólo existen 256 en definitiva), las diferentes funciones se asociaron a diferentes interrupciones, según sea el hardware que controlan.

La tabla 3.1 muestra las diferentes interrupciones. Además, el BIOS utiliza algunas interrupciones como variables, que por ejemplo aceptan punteros a tablas de vídeo o de disco duro. Para que el BIOS pueda separar las diferentes funciones, a cada función se le asocia un número de función, que se ha de pasar al BIOS en el registro AH del procesador, cuando se llama la interrupción. Y este registro no es el único que se ha de cargar al llamar a la interrupción, ya que todo el paso de parámetros se desarrolla mediante la ayuda de los registros del procesador. Cómo se utilizan los registros del procesador en este proceso también es una parte del estándar BIOS y se define de forma totalmente individual para cada función.

Interr.	Servicios
10h	Acceso a la tarjeta de vídeo
11h	Determinar la configuración
12h	Determinar el tamaño de memoria RAM
13h	Funciones de disquetes/disco duro
14h	Puerto serie
15h	Funciones de cassete y funciones ampliadas de AT
16h	Teclado
17h	Puerto paralelo
18h	Fecha/Hora y reloj de tiempo real mantenido por baterías

Tabla 3.1 Las interrupciones del ROM-BIOS

3.1.3 El puerto serie

Al abandonar el puerto serie, los bits entran en un mundo regido por la norma RS-232. Esta norma determina tanto las dimensiones del conector como el número de clavijas y su posición, o los diferentes parámetros eléctricos. Para decirlo exactamente, en el mundo de la PC cuando se habla de puerto serie nos referimos irremediablemente a RS-232, se

revesirá con más detalle este punto, en este capítulo cuando se toque la referente a la implementación de los módulos de transmisión inalámbrica.

En el interior del puerto serie hay un chip especial para la entrada y salida de caracteres y, sobre todo, para la conversión de palabras de datos en las correspondientes señales del puerto serie: lo que se denomina UART. Estas siglas corresponden a Universal Asynchronus Receiver Transmitter, es decir, algo así como emisor y receptor asíncrono universal, si bien cada vez más uno se topa con el nombre SIO, abreviatura de Serial Input/Output.

Concentrémonos ahora en la estructura, modo de funcionamiento y los registros de este chip.

3.1.3.1 Los registros del UART

El UART dispone en total de diez registros accesibles desde el exterior (vía software). Además, dispone de algunos otros registros adicionales accesibles sólo internamente. Entre ellos podemos contar por ejemplo el Receiver Shift Register y el Transmitter Shift Register, que juegan un papel fundamental en la emisión y recepción de caracteres. Si el UART recibe un carácter, los diferentes bits que se van recibiendo se amontonan primero en el Receiver Shift Register hasta que se completa una palabra de datos. Si no aparece ningún fallo, el Byte es transferido al Receiver Data Register desde donde puede ser leído vía software.

En el sentido contrario, el software escribe primero la palabra de datos a enviar en el Transmitter Holding Register. De ahí el UART lo traslada al registro interno Transmitter Shift Register para que desde ahí transmitir los diferentes bits uno detrás de otro a través de la línea. Es por este motivo que son tan importantes para la transmisión de datos vía UART tanto el Transmitter Holding Register como el Receiver Data Register. No hay que olvidar sin embargo el resto de registros, pues tienen funciones tan necesarias como la inicialización del UART o la consulta del estado de la transmisión y de la línea. Véase la tabla 3.2.

Transmitter Holding	THR	0	0
Receiver Data	RBR	0	0
Baudrate Divisor LSB	DLL	0	1
Baudrate Divisor MSB	DLM	1	1
Interrupt Enable	IER	1	0
Interrupt ID	IIR	2	-
Line Control	LCR	3	-
Modem Control	MCR	4	-
Line Status	LSR	5	-
Modem Status	MSR	6	-

Tabla 3.2: Los registros internos del UART 8250

3.1.3.2 Acceso a los registros

Los registros del UART son accesibles desde diferentes puertos que se orientan en la dirección de base del puerto serie. Esta dirección de base puede, teóricamente, escogerse a voluntad, pero en la práctica los dos primeros puertos serie de un PC (COM1 y COM2) son accesibles a través de las direcciones de base 3F8h y 2F8h. Para COM3 y COM4 (si están presentes) normalmente se utilizan las direcciones de base 3E8h y 2E8h. Si se quiere tener la certeza de que, incluso en circunstancias especiales, no se pasará de largo el puerto serie, lo mejor es no fijar las direcciones de puerto desde un principio en los programas. En lugar de esto se recomienda consultar una de las cuatro variables del BIOS en las que se encuentran las direcciones de base de los como máximo cuatro puertos serie que puede soportar el BIOS. De esta manera se puede obtener con seguridad la dirección actual del puerto serie deseado. Véase la tabla 3.3.

Puerto	Dirección en BIOS - Variable	Puerto estándar
COM1	0040:0000	3F8h-3FFh
COM2	0040:0002	2F8h-2FFh
COM3	0040:0004	3E8h-3EFh
COM4	0040:0006	2E8h-2EFh

Tabla 3.3: Las direcciones de base de los diferentes puertos serie y las variables del BIOS en las que se encuentran

Si vuelve a repasar la tabla con los diferentes registros del UART se percatará de que dos direcciones de puerto (puerto base +0 y puerto base +1) son ocupadas por varios registros. Para poder diferenciar entre los distintos registros al acceder al puerto correspondiente se utiliza el bit más significativo en el Line Control Register.
Antes de acceder a uno de los registros del primer o segundo puerto serie, cargue este bit con el valor indicado en la tabla.

3.1.3.3 El estado actual del UART

Las informaciones de estado más importantes sobre la emisión y recepción de caracteres y el estado de la línea de transmisión se pueden consultar a través del Line Status Register (LSR).

El bit 0 muestra si un carácter recibido se encuentra en el registro de recepción (RBR). Este bit se inicializa automáticamente tan pronto como el carácter recibido es leído del registro. Si esto no sucede de manera suficientemente rápida, puede suceder que bajo ciertas circunstancias haya un nuevo carácter preparado antes de que el último haya podido ser leído. En este caso, el carácter antiguo sencillamente sería sobrescrito y se perdería irremediablemente. Esto quedaría indicado a través del bit 1 del LSR.

Los bits 2 y 3 se utilizan para indicar dos errores más: errores de paridad y los denominados Overrun Errors. Estos últimos aparecen cuando no se respeta el protocolo, en

lo que se refiere al número de bits de datos, stop bit o bit de paridad en la recepción de un carácter, normalmente debido a un fallo de la línea.

El UART muestra todos los errores pero sin tomar medidas al respecto, pues corresponde al software de comunicación desarrollar un protocolo y unos mecanismos capaces de avisar a emisor y receptor sobre transmisiones defectuosas y provocar la repetición de la transmisión de los datos correspondientes.

Cuando el emisor activa una alarma a través del bit 6 del Line Control Register, ésta se refleja en el bit 4 del Line Status Register del receptor con lo que éste puede reaccionar inmediatamente.

Cierta información sobre el estado actual de la emisión la suministran los bits 5 y 6 de LSR, que aluden a los dos registros de emisión, el Transmitter Holding Register y el Transmitter Shift Register. El bit 5 indica si el Transmitter Holding Register está vacío. Si es así, el siguiente carácter a enviar debe cargarse en este registro. En caso contrario, el software debe esperar hasta que este bit se pone a cero.

Por el contrario, la información de si el Transmitter Shift Register está vacío o no es menos importante. Si está vacío, significa que el último carácter a enviar ya ha sido completamente transferido. Sino, éste todavía se encuentra en el TSR y por tanto no ha llegado aún al receptor.

3.1.4 Acceso al puerto serie vía BIOS

Dado que el puerto serie pertenece a la lista de componentes básicos de un PC, el BIOS no lo deja desatendido. Sin embargo, el soporte al programador es muy reducido. Hay cuatro funciones disponibles que únicamente permiten la configuración de los parámetros de la transmisión, la consulta del estado de la línea y el envío y recepción de caracteres en modo de verificación constante[3]. Faltan medios de ayuda para el funcionamiento controlado por interrupciones del puerto serie, así como funciones para el acceso a las posibilidades ampliadas del 16550 y sus sucesores. En la práctica uno se ve obligado a programar directamente los diferentes registros del puerto, tal como se ha visto anteriormente. Por eso explicaremos únicamente las funciones del BIOS para conservar la integridad del texto.

Número de función	Cometido
00h	Configuración de los parámetros de la comunicación
01h	Enviar caracteres
02h	Leer caracteres
03h	Consultar el estado del puerto

Tabla 3.4: Las funciones del BIOS para el acceso al puerto serie

[3] El término utilizado en ingles es polling

Todas las funciones para el acceso al puerto serie se llaman con la interrupción 14h, indicando el número de la función deseada en el registro AH. Además, en el registro DX se espera el número del puerto al que se desea acceder, donde 0 representa COM1, 1 COM2, y así sucesivamente. Véase la tabla 3.4.

3.1.4.1 Configuración de los parámetros de comunicación y consulta de estado

Junto a los argumentos AH y DX anteriormente mencionados, para la configuración de los parámetros de la comunicación se ha de suministrar a la función 00h en el registro AL un valor cuya estructura se refleja en la siguiente figura. En ella se observa que el BIOS sólo admite un subconjunto de las posibles configuraciones directas que permite el UART. Así, por ejemplo, sólo se puede configurar una longitud de palabra de 7 u 8 bits, e incluso el espectro de los posibles ratios de baudios está muy por debajo de las posibilidades que ofrece el UART.

Como valor de respuesta la función 00h devuelve al solicitante en el registro AH el estado de la línea que ha obtenido del Line Status Register del UART y en el registro AL, el estado del módem, obtenido del Módem Status Register.

El estado de la línea, tal como lo devuelve el registro AH, se puede consultar cuando se quiera con ayuda de la función 03h. Al llamar a esta función no se espera ningún otro argumento aparte del número de función en AH y el número de puerto en DX. El estado del puerto se suministra como está representado más arriba, en el registro AH.

3.1.4.2 Envío y recepción de caracteres

Para enviar caracteres se utiliza la función 01h. Además del número de función y el número de puerto, al llamarla se debe cargar el carácter a enviar en el registro AL. Si el carácter es enviado sin problemas, tras la llamada a la función el bit 7 del registro AH se pone a cero. Un uno indica que el carácter no se pudo transmitir. El resto de bits corresponden al estado de la línea.

Para recibir un carácter debe recurrirse a la función 02h. Si se recibe un carácter, tras la llamada a esta función, el registro AL contiene el carácter recibido. Si no se ha producido ningún fallo, el registro AH contiene un cero, de lo contrario su valor se corresponde con el del estado de la línea.

3.1.4.3 Verificación constate o interrupción

La comunicación entre el software y el UART puede darse tanto en modo de verificación constante como en modo interrupción. En modo de verificación constante, es responsabilidad del software consultar a través del Line Control Register el estado del UART a espacios de tiempo regulares. Sólo así puede determinar si se ha recibido un nuevo carácter o si el último carácter enviado se encuentra realmente en camino. Los correspondientes módulos de programas son sencillos de construir, pero resucitan con ellos la principal desventaja de todos los procedimientos de verificación constante: la CPU está todo el tiempo ocupada con el dispositivo, aunque los caracteres, comparándolo con la velocidad de la CPU, son enviados y recibidos de manera muy lenta.

Así, dependiendo de la tarea a realizar, normalmente se prefiere trabajar con el procedimiento de interrupciones, que si bien es un poco más costoso de programar, ofrece sin embargo todas las ventajas conocidas de la manipulación de interrupciones: el CPU sólo tiene que utilizar el puerto serie cuando realmente llega un carácter, tiene que enviarse o debe sortearse un fallo, pues sólo en estas situaciones inicia el UART una interrupción y activa con ello el administrador de interrupciones. El UART soporta este modo de trabajo a través de dos registros de interrupción, de los que hablaremos posteriormente.

A continuación se hace una descripción detallada de las partes importantes del código fuente del programa encargado de la administración y manejo de la interfáz de usuario que se usará para el envío y recepción de información a través del puerto serie.

3.1.5 El programa de transmisión y recepción

Para el acceso al puerto serie el programa utiliza la rutina del módulo auxiliar SERUTIL, Como muestra la siguiente lista, en este módulo se encuentran todas las rutinas necesarias para acceder a los diferentes registros del UART, para mostrar el estado del UART en pantalla, para consultar el tipo de UART y muchas cosas más. Véase la tabla 3.5.

Función	Cometido
ser_UARTType	Averiguar el tipo de chip del UART
ser_Init	Inicializar el puerto serie
ser_FIFOLevel	Fija el tamaño del buffer FIFO
ser_IsDataAvailable	¿Hay datos preparados para ser leídos?
ser_IsWritingPossible	¿Puede el puerto enviar el siguiente Byte?
ser_WriteByte	Enviar un Byte
ser_ReadByte	Recibir un Byte
ser_WritePaket	Enviar un paquete de datos
ser_ReadPaket	Recibir un paquete de datos
ser_CLRIRQ	Interrumpir un mensaje de una interrupción serie al controlador IRQ
ser_SETIRQ	Autorizar un mensaje de una interrupción serie al controlador IRQ
ser_SetIRQHandler	Activar el administrador de interrupción
ser_RestoreIRQHandler	Restaurar el antiguo administrador de interrupción
ser_PrintError	Emitir mensaje de error
ser_PrintModemStatus	Mostrar el estado de la línea de señal
ser_PrintLineStatus	Mostrar el estado del puerto
ser_GetBaud	Averiguar el ratio de baudios actual de un puerto
ser_PrintCardSettings	Mostrar los parámetros de transmisión del puerto

Tabla 3.5: Las funciones del módulo SERUTIL

3.1.5.1 Inicialización del UART

La primera parte del programa se encarga de las configuraciones necesarias para el establecimiento de la comunicación. Para establecer conexión con otro puerto serie, el UART primero debe ser inicializado. Especialmente en lo que concierne a los diferentes parámetros de comunicación, es decir, la tasa de baudios, la longitud de la palabra de datos y el número de stop bit. Una buena práctica consiste también en llevar a cabo un acceso de lectura al Receiver Data Register. Si algún programa anterior ha dejado ahí algún carácter, existe el peligro de que se interprete por error este carácter como el primero de la conexión que se está a punto de establecer. Si, por el contrario, no existe ningún carácter preparado para ser llamado, el acceso de lectura no tiene ningún efecto por lo que no puede cometerse ningún error.

Lo importante, al configurar los diferentes parámetros de la comunicación, es empezar por la tasa de baudios pues al escribir los distintos registros, el UART inicializa el resto de parámetros. Así, si se configuran estos registros antes de indicar la tasa de baudios, tendrán que volverse a configurar de nuevo.
Para la determinación de la razón de baudios existen los registros DLL y DLM. El valor deseado no se entra directamente, sino como un cociente con respecto a la frecuencia del UART, que es de 1,8432 MHz. La razón de baudios comunica al UART que tan rápido debe generar los diferentes bits con respecto a su frecuencia de reloj. En concreto, la fórmula sería:

Valor del registro = 1,8432 MHz / (16 * ratio de baudios)

La igualdad expresa que el UART debe dividir la duración de un pulso por 16 y tras N (con N = ratio de baudios) de estos pulsos, enviar el siguiente bit a la línea.
Como valor del registro se obtiene, con esta fórmula, un valor de 16 bits, cuyo Byte menos significativo debe escribirse en DLL y cuyo Byte más significativo debe escribirse en DLM. Teóricamente, procediendo de esta manera se puede escoger cualquier ratio de baudios comprendido entre 1,75 baudios (valor del registro = 0FFFFh) y 115200 baudios (valor del registro = 1). En la práctica, sin embargo, se opera con unos valores muy concretos, que se relacionan en la siguiente tabla. En ella también se indican los valores de registro que deben introducirse en DLL y DLM para obtener la razón de baudios deseado.

La velocidad de transmisión más elevada que puede conseguirse es 115200 baudios, es decir 115 Kilobits por segundo. Recuérdese que por cada palabra de datos de 8 bits deben transmitirse un mínimo de dos bits adicionales (1 bit de inicio y un stop bit), con lo que la razón de transmisión de datos real queda en 1,4 KByte por segundo. De todos modos, este elevado ratio de transferencia sólo puede conseguirse en la comunicación directa entre dos PC's mínimamente actuales. Por debajo de un 386 a 16 MHz resultará difícil obtener tan altos índices de transferencia. Véase la tabla 3.6.

En el programa de la interfaz de usuario, la velocidad de transmisión se tiene que expresar como uno de dos argumentos posibles al momento de hacer el llamado del

programa en la línea de comandos. Si no se especifica la velocidad de transmisión s, por defecto se asigna la velocidad de 9600 baudios. Además se especifica el número de puerto serie de comunicación, ya sea el puerto 1 ó 2, si no se especifica, se toma el puerto 1 por defecto. Si dentro de los argumento se encuentra el carácter '?', el programa despliega la sintaxis que se debe utilizar para la ejecución del programa y termina el programa. El procedimiento *ser_init* se encarga de inicializar el puerto serie, este procedimiento recibe tres parámetros, el primero el es puerto que se desea inicializar , el segundo es la velocidad de transmisión y por último una máscara de bits que se encarga del resto de los parámetros, como la longitud de palabra, el número de stop bit y la utilización de bits de paridad se configuran a través del Line Control Register (LCR) que se encuentra en la dirección relativa 3 con respecto a la dirección de base del puerto serie. Este registro se puede escribir para llevar a cabo nuevas reconfiguraciones, pero también se puede leer si se quiere averiguar la configuración actual.

Ratio de baudios	Divisor	Registro DLM	Registro DLL
50	2304	09h	00
75	1536	06h	00h
110	1047	04h	17h
134,5	857	03h	59h
150	768	03h	00h
300	384	01h	80h
600	192	00h	C0h
1200	96	00h	60h
1800	64	00h	40h
2000	58	00h	3Ah
2400	48	00h	30h
4800	24	00h	18h
7200	16	00h	10h
9600	12	00h	0Ch
19200	6	00h	06h
38400	3	00h	03h
57600	2	00h	02h
115200	1	00h	01h

Tabla 3.6: Velocidades en Baudaje más usuales y los valores a introducir en los registros DIM y DLL del UART para conseguirlos.

```
if( FindString( argv, "?", argc ) )
```

```
  /* Busca en la cadena introducida en la linea de comandos al   hacer el llamado
del programa y despliega ayuda de sintaxis cuando el usuario coloca un signo ? en la
cadena*/  (
    printf("Sintaxis:\n" );
    printf("SERIRQ [-COM:comport] [-BAUD:baudrate]\n");
    printf("comport  = 1 ¢ 2       (Por defecto: 1)\n");
    printf("baudrate = 50 - 115200 (Por defecto: 9600)\n");
    exit(0);    /* Termina la ejecución del programa  */
  )
  if( GetArg( argc, argv, "-COM:", _int, &iCom, 1 ) )
  {   /*si el puerto especificado es el puerto 1*/
    if( iCom == 1 )
    {      /*si el puerto especificado es el puerto 1*/
        iSerPort = SER_COM1;    /* Sólo COM1 y COM2 están  "estandarizados" */
        iSerIRQ  = SER_IRQ_COM1;
    }
    else
    if( iCom == 2 )
    {     /*si el puerto especificado es el puerto 2*/
        iSerPort = SER_COM2;    /* Sólo COM1 y COM2 están  "estandarizados" */
        iSerIRQ  = SER_IRQ_COM2;
    }
    else
    {   /*si se especifica cualquier otro puerto se termina el programa*/
        printf("Puerto COM no soportado\n");
        exit(0);
    }
  }
  else
  {     /* Si no se especifica puerto, se toma el COM1 por defecto */
    iSerPort = SER_COM1;    /* Sólo COM1 y COM2 están  "estandarizados" */
    iSerIRQ  = SER_IRQ_COM1;
  }
  if( !GetArg( argc, argv, "-BAUD:", _long, &lBaud, 1 ) )
        lBaud = 9600L;                /* Tasa máxima en UART 8450A */
  if( lBaud > SER_MAXBAUD )
  {   /* Si la velocidad de transmisión es mayor a SER_MAXBAUD =115200,
         termina el programa */
    printf("Baudrate demasiado alto\n");
    printf("M ximo: %ld Bd\n", SER_MAXBAUD );
  )
/*inicialización de la UART*/
 iUART = ser_Init( iSerPort, lBaud,
                   SER_LCR_8BITS | SER_LCR_1STOPBIT | SER_LCR_NOPARITY );
  if( iUART == 10000 )
  {
    printf("No hay puerto\n");
    exit( 0 );
  )
```

3.1.5.2 Buffer para la recepción y la transmisión

Un punto muy importante a considerar es la configuración del buffer FIFO del UART. La falta de unos buffer, habría limitado mucho el funcionamiento del UART en velocidades de transmisión elevadas, sobre todo en lo que hace referencia al modo de funcionamiento por interrupciones. Raramente se llegaba a valores por encima de los 9600 baudios, pues a menudo el chip recibía los caracteres más rápidamente de lo que era capaz de transmitirlos al software vía interrupciones. Por un lado, porque las computadoras no eran suficientemente veloces y, por el otro, porque las solicitudes de interrupción del UART no se podían dar inmediatamente. Además, el puerto serie sintonizado

automáticamente con IRQ3 o IRQ4 no disponían ni de lejos la mayor prioridad de interrupción dentro del sistema.

Como consecuencia, era frecuente que un carácter recién llegado al Receiver Data Register fuera eliminado por la sobre escritura del siguiente carácter dándose así un error de sobrecarga de memoria. En estos casos tenía que reducirse la razón de baudios, de modo que funcionase la comunicación entre emisor y receptor.

En el NS16550A y su sucesor, el NS16C552, este problema ha perdido importancia de una manera sustancial pues disponen de sendos buffer de 16 Byte intercalados antes del Receiver Buffer Register (recepción) y el Transmitter Holding Register (envío). Ambos buffer trabajan bajo el principio FIFO (first in first out) por lo que han dado sobrenombre al chip 16550, que frecuentemente se conoce sencillamente como FIFO.

El procedimiento *ser_FIFOLevel* se encarga de fijar el tamaño del buffer FIFO, recibe como parámetro el puerto a configurar y el tamaño del buffer FIFO. Observe la sección de código que implementa esta funcionalidad

```
if( iUART > INS8250 ) ser_FIFOLevel( iSerPort, SER_FIFO_TRIGGER14 );
```

De los dos buffer, el de envío juega un papel más bien secundario pues la PC, al enviar, puede determinar por sí mismo la cadencia del envío, con lo que apenas existe la posibilidad de autoexigirse en exceso. Por el contrario, el buffer de recepción representa una gran ayuda, aunque el software tenga que jugar un poco con él para sacarle provecho. Nos referimos por un lado a la inicialización del UART y, por otro, a la estructuración de la rutina para la lectura de caracteres.

En el modo de funcionamiento por verificación constante, las cosas no cambian mucho, pues normalmente se consulta primero el Line Status Register para saber si hay algún carácter disponible en el RBR. Si es así, se lee el carácter y a continuación se inicia el proceso de nuevo. Si todavía no hay caracteres preparados en el buffer interno, el UART, inmediatamente después de la lectura del carácter del RBR, activará de nuevo el bit correspondiente del Line Status Register de modo que este carácter sea automáticamente recogido en el próximo bucle.

Es precisamente un bucle de este tipo lo primero que falta en la mayoría de administradores de interrupciones de puertos serie, pues, mientras el UART no tiene ningún carácter en un buffer interno, es suficiente con leer una vez el RBR al llamar a la interrupción y entonces esperar a la siguiente llamada a una interrupción para recibir el siguiente carácter. Al instalar el buffer tiene que implementarse en cambio en el administrador de interrupciones un bucle en cuyo proceso continuamente se lee un carácter del RBR hasta que el Line Status Register indica que ya no hay más caracteres en el RBR para ser leídos. En este momento se han leído todos los caracteres del buffer interno pues por su parte el UART habría empujado el resto de caracteres que todavía se encontrasen en el buffer, tras la lectura de sus precedentes, hacia el RBR.

Para continuar siendo compatible con sus antecesores, el 16550 desactiva su buffer interno mientras no se le diga explícitamente lo contrario. Para ello existe un nuevo registro que se encuentra en la dirección 2 relativa a la dirección de base del puerto. Si vuelve a echar una mirada atrás y repasa la tabla 3.2 en la que se encuentran los registros del puerto serie, podrá observar que en esta dirección se encuentra un registro, el Interrupt ID Register. Mientras que el Interrupt ID Register sólo puede ser leído, el nuevo registro del

16550 para el funcionamiento en FIFO sólo puede ser escrito. Con ello es posible dividir la dirección entre uno y otro registro con lo que en función del tipo de acceso, el UART diferencia a qué registro se quiere acceder.

A través del bit 0 de este registro se activan los buffer tipo FIFO. Si en algún momento se necesita borrar uno de los dos buffer, basta con escribir el valor 1 en el bit 1 (buffer de recepción) o en el bit 2 (buffer de emisión). Para ello no olvide poner también el bit 0 pues de lo contrario el buffer FIFO se activaría simultáneamente.

A través de los bits 6 y 7 se puede configurar el número de caracteres a partir de cuya recepción se iniciará una interrupción siempre que el UART esté configurado para generar interrupciones a través del Interrupt Enable Register. Escogiendo los valores 4, 8 o 14 en estos bits se motivará que el UART no envíe una interrupción cada vez que reciba un carácter. Esto influye muy positivamente en las prestaciones del software pues, en términos de medida de procesador, la ejecución de una interrupción conlleva mucho tiempo, por lo que toda reducción del número de éstas es positiva. El bucle de consulta explicado anteriormente dentro de la rutina de la interrupción se ocupa de que realmente se lean todos los caracteres del buffer y no sólo el primero.

Sin embargo este procedimiento tiene un problema decisivo. ¿Qué sucede cuando el lado opuesto envía tres caracteres configurando con ello por primera vez la transmisión, porque estos tres caracteres le dicen al receptor ahora me devuelves los caracteres y la información? En ciertas circunstancias puede suceder que al receptor no le lleguen estos tres caracteres porque el buffer todavía no tiene los cuatro, ocho o catorce caracteres mínimos necesarios antes de que sea lanzada una interrupción. Con ello, el emisor podría estar esperando eternamente una respuesta del receptor.

Para que no se dé este caso el UART provoca siempre una interrupción cuando el buffer no está lleno pero ha transcurrido el tiempo suficiente para la transmisión de tres caracteres y no se ha recibido ninguno. Este Time Out le es indicado al administrador de interrupciones a través de un nuevo bit adicional en el Interrupt ID Register. Este bit básicamente puede ser ignorado pues el UART continúa teniendo como argumento para lanzar una interrupción el que haya caracteres disponibles en el RBR.

Si se quiere desarrollar un programa para la comunicación a través del puerto serie, debe incluir siempre las posibilidades del FIFO, siempre que esté disponible.

3.1.5.3 Controlador o administrador de interrupciones

El paso siguiente es escribir el *controlador o administrador de interrupciones*. Si la UART debe ser alimentada con caracteres vía interrupciones, es decir, si la lectura de los caracteres que van llegando se realiza mediante la ayuda de las interrupciones, no pueden prescindirse de los dos registros de interrupción. Uno, el Interrupt Enable Register (IER) determina las situaciones en que las interrupciones deben ser llevadas a cabo. El otro, el denominado Interrupt Identification Register (IIR), indica al administrador de interrupciones del puerto serie al llamarlo con qué objeto se le ha requerido.

Los administradores de interrupciones debe estar disponible vía software. El programa muestra cómo construir uno de estos administradores de interrupción. Para que el administrador pueda ser llamado correctamente, su dirección debe estar contenida en el vector de interrupciones existente para los puertos serie. Para el primer puerto serie es

IRQ4 con el vector de interrupción 0Ch, y para el segundo, IRQ3 con el vector de interrupción 0Bh. Con el tercer y cuarto puertos serie, si están presentes, la correspondencia no es tan clara pues hay diferentes posibilidades de ocupación tanto en lo que se refiere a la dirección del puerto como a la interrupción utilizada. Para la instalación del administrador de interrupciones se recurre al procedimiento llamado *ser_SetIRQHandler*. Este procedimiento recibe como parámetros el puerto serie para el que se requiere instalar un administrador de interrupciones, el número de interrupción que esta asignado al puerto, la dirección del nuevo administrador de interrupciones y estados que se pueden provocar por una interrupción, además regresa la dirección del antiguo administrador, vea el llamado a esta rutina

```
  /* Instalar el controlador de interrupciones, la variable lpOldIRQ guarda la
dirección
     del controlador IRQ antiguo  */
  lpOldIRQ = ser_SetIRQHandler( iSerPort,       /* instalar interrupción */
                                                iSerIRQ,        /* serie
*/
                                                GetSer,
                                                SER_IER_RECEIVED | SER_IER_SENT );
```

Además de tener lo necesario para la instalación del administrador de interrupciones, nos introducimos ahora a la construcción del propio administrador. Para la construcción del administrador de interrupciones se introduce un modificador de funciones en el lenguaje C++, este modificador es necesario para el manejo de interrupciones. Este modificador es el modificador *interrupt*. La palabra reservada interrupt de C++ define una función como manejador de interrupciones. Una función interrupt es invocada directamente por el hardware vía el mecanismo de manejo de interrupciones del 80x86. Las funciones ordinarias no deben declararse como interrupt. Además, las funciones invocadas por funciones interrupt no deben declararse como interrupt por si mismas.

Una función interrupt no puede ser invocada con argumentos y tan poco puede devolver nada a quien la llama. El origen de la llamada es un mecanismo que forma parte del hardware en el que, en general, no se puede manejar los valores devueltos por la función. Normalmente una función interrupt debe ser instalada antes de que pueda utilizarse.

La función interrupt maneja transparentemente todos los detalles del almacenamiento y restauración de registros. Cuando ocurren interrupciones, sean de software o hardware, el CPU busca en su tabla de excepciones la dirección de una función a la cual llama para que se haga cargo de la interrupción. Como las interrupciones son asíncronas con respecto al programa principal, las rutinas de servicio de interrupciones necesitan preservar el contenido de todos los registros de la máquina. La palabra reservada interrupt produce los siguientes cambios en el código compilado:

1. Todos los registros se almacenan en la entrada.
2. Todos los registros se restauran a la salida.
3. Para salir se utiliza una instrucción iret.

Considere el siguiente ejemplo.

```
void interrupt alguna_funcion()
{
        // Cuerpo de la función
}
```

Esta función genera el siguiente código compilado.

```
alguna_funcion:
                        ;salva todos los registros
push ax
push bx
push cx
push dx
push es
push ds
push si
push di
push bp

---- cuerpo de la función
                        ; restaurar todos los registros
pop bp
pop di
pop si
pop ds
pop es
pop dx
pop cx
pop bx
pop ax
iret
```

Aunque C++ permite siempre devolver un valor de una función interrupt realmente no tiene sentido hacerlo. Intentar devolver un valor de una función interrupt es tan solo pérdida de tiempo, ya que C++ utiliza el registro ax para devolver valores, y el registro es restaurado con su valor original al momento de la salida.

En el programa de transmisión, se define precisamente una función interrupt para el administrador de interrupciones, vea el detalle de la implementación del administrador de interrupciones:

```
/* GetSer : Rutina de interrupción disparada por el puerto serie              */
/*: En esta función se reciben datos del puerto serie, y se
*/
/*        representan en la ventana 'Remote'. Esta función contiene          */
/*        además el esqueleto para todos los demás eventos de puerto   */
/*        serie, que disparen una IRQ.
*/
```

```
VOID __interrupt _FP GetSer( VOID )
{ static BYTE bIRQID;
  static int i,pot;
  static long t;
  bIRQID = ( BYTE )inp( iSerPort + SER_IRQ_ID );
  if( ! ( bIRQID & SER_ID_PENDING ) )                    /* IRQ esperando ? */
  {
    switch( bIRQID & SER_ID_MASK )
    {
 case SER_ID_RECEIVED:                          /* Después de recepci¢n */
```

... Manejo de la interrupción ...

```
        break;
        case SER_ID_SENT:                               /* Después de envío  */
        break;
        case SER_ID_MODEMSTATUS:  /* Después de modif. de estados de línea */
        break;
    }
  }
  irq_SendEOI( iSerIRQ );                              /* final de la interrupción */
}
```

Una vez que se dispara la interrupción, se llama a la función GetSer , en esta función se implementa toda la lógica y el tratamiento de la información que se recibe en el puerto serie, incluyendo la decodificación de viterbi de los datos. Comienza por verificar el número de interrupción que disparó la función, si se trata del número de interrupción correspondiente al puerto serie, entonces comienza a verificar qué tipo de acontecimiento relacionado con el puerto serie sucedió, si ocurrió una transmisión o u cambio de estado de módem o una recepción. Este último caso es el que nos interesa, y en él se implementa toda la lógica de procesamiento para el manejo de la interrupción, tal como se muestra en la sección de código que se acaba de mostrar.

Es aquí cuando comenzaremos el análisis de la decodificación de la información que se recibe en el puerto serie. Primero mencionaremos que el envío de los datos está diseñada de tal manera que al enviar la unidad mínima de información, constituida por un byte, se pasa a través de un proceso de codificación convolucional y esto nos da como resultado que por cada byte que se procesa vamos a tener 3 bytes codificados para ser puestos en el canal (más adelante revisaremos este proceso) tal como lo ilustra la siguiente gráfica.

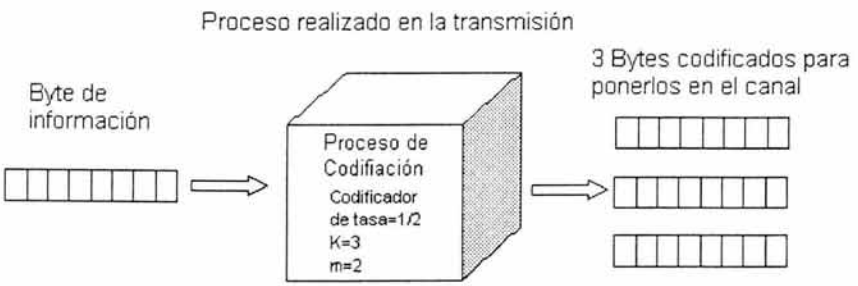


Figura 3.1 En el proceso de transmisión, se codifican los bytes antes de ponerlos en el canal, debido a la redundancia que se agrega al byte se generan tres bytes por byte de información.

entonces, lo primero que se debe hacer al recuperar los bytes que se están recibiendo en el puerto serie del receptor es guardarlos en un arreglo, en la implementación del administrador del receptor existe un arrelgo llamado *megrec* en donde se guardan los bytes que se van recibiendo a través del puerto serie. El índice k del arreglo megrec, sirve también como contador para saber el momento en que se han recibido ya los tres bytes que corresponden a la codificación de 1 bit de información. Observe la siguiente ilustración

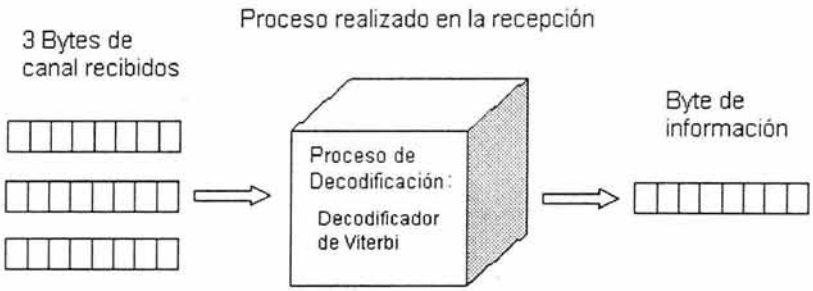


Figura 3.2 En el proceso de recepción, se aplica la decodificación de Viterbi a los bytes de canal que se recibe.

es entonces cuando se hace una transferencia a dos arreglos, llamados *encodedr* y *encodedrl*, cada uno contendrá 12 del total de bits contenido en los 3 bytes de canal almacenados en el arreglo msgrec. La transferencia a los dos arreglos se hace con la finalidad de que estos arreglos sirvan de entrada para el proceso de decodificación de viterbi, este proceso de decodificación es desarrollado por el procedimiento *sdvd* que recibe como parámetro de entrada, un arreglo de enteros. Para realizar la transferencia de los bits a los arreglos, tomamos en consideración el siguiente hecho. Un byte es una colección de 1's y 0's cada uno con un peso relativo de acuerdo a su posición en el byte, esto es

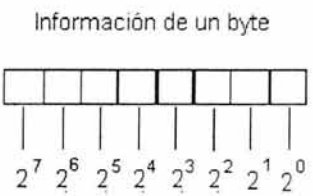


Figura 3.3 Peso binario de los bits dentro de byte debido a su posición.

y para extraer un bit en particular dentro de un byte de información, lo que hacemos es aplicar el aperador AND entre el byte de información con un byte máscara, cabe recordar que el operador AND opera bit a bit. El byte máscara es un byte cuyo valor es el peso

binario que corresponde al bit que se quiere extraer ,es decir, si se quiere extraer el bit menos significativo (el bit más a la derecha), el byte máscara será igual al peso binario 2^0 = 1, si se quiere obtener el siguiente bit, el byte máscara será igual al peso binario 2^1=2, asi sucesivamente hasta obtener el bit más significativo, con un byte de máscara igual al peso binario 2^7 = 128, veamos este último caso gráficamente.

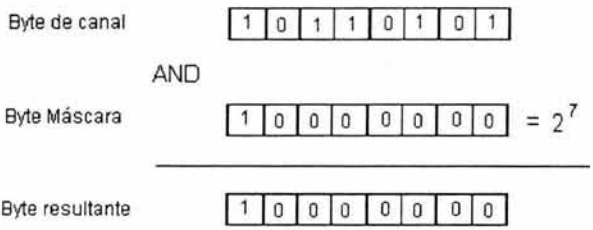


Figura 3.4 Ejemplo de aplicar el operador AND bit a bit entre un byte de canal y un byte "Mascara"

observamos que el byte máscara tiene los bits igual a 0 para todas las posiciones excepto para la posición de la cual queremos obtener el bit, esto asegura que al aplicar el operador AND en estas posiciones vamos a obtener 0, pero para la posición de la que queremos obtener el bit tenemos el valor de 1, el resultado de la operación AND bit a bit será 1 por lo que el byte resultante será diferente de cero, de lo contrario, si el bit de la posición deseada es cero, el resultado de la operación AND bit a bit será 0 por lo que el byte resultante será cero. Este mecanismo nos permite leer cualquier bit dentro de un byte que se recibe a través de un puerto serie. De forma particular, para la implementación del administrador se implementaron 3 ciclos que corresponden a los tres bytes resultado de la codificación convolucional de un byte del mensaje original. Veamos la implementación de este proceso dentro del administrador de la interrupción.

```
/* GetSer : Rutina de interrupción disparada por el puerto serie              */
/*: En esta función se reciben datos del puerto serie, y se                   */
/*        representan en la ventana 'Remote'. Esta función contiene          */
/*        además el esqueleto para todos los demás eventos de puerto   */
/*        serie, que disparen una IRQ.
*/
VOID __interrupt _FP GetSer( VOID )
{ static BYTE bIRQID;
  static int i,pot;
  static long t;
  bIRQID = ( BYTE )inp( iSerPort + SER_IRQ_ID );
  if( ! ( bIRQID & SER_ID_PENDING ) )              /* IRQ esperando ? */
  {
    switch( bIRQID & SER_ID_MASK )
    {
        case SER_ID_RECEIVED:                      /* Después de recepción */
       i = 0;
/* Mientra haya datos disponibles en el puerto serie, se leen y se guardan en el
arreglo  msgrec */
       while( ser_IsDataAvaiable( iSerPort ) && ( i < 16 ) )
       {
         msgrec[ k++ ] = ( BYTE )inp( iSerPort + SER_RXBUFFER );
         msgrec[ k ] = '\0';
       }
if (k>2)          /*  Si se han recibido los tres bytes, que corresponden a la
codificación de un solo bit */
    {    k=0;
```

```
        indice=0;
        for (t = 0; t < 8 ; t++)      /*guardar en el arreglo el contenido del primer
byte  */
        {
          /* obtiene el peso binario */
          pot=(int)pow(2,(7-t));
          /* aplica operación AND entre el byte del mensaje y el byte máscara*/
          if ((msgrec[0] & pot)!=0)
               *( encodedr + t ) = -1;
          else
               *( encodedr + t ) = 1;
        }
        for (t = 0; t < 8 ; t++)     /*guardar en el arreglo el contenido del segundo
byte  */
        {
          pot=(int)pow(2,(7-t));
          if ((msgrec[1] & pot)!=0)
               if (t<4)
                  *( encodedr + (t+8)) = -1;
               else
                  *( encodedr1 + (t-4)) = -1;
          else
               if (t<4)
                  *( encodedr + (t+8)) = 1;
               else
                  *( encodedr1 + (t-4)) = 1;
        }

        for (t = 0; t < 8 ; t++)    /*guardar en el arreglo el contenido del tercer
byte  */
        {
          pot=(int)pow(2,(7-t));
          if ((msgrec[2] & pot)!=0)
               *( encodedr1 + (t+4)) = -1;
          else
               *( encodedr1 + (t+4)) = 1;
        }
                  /* aquí se hace la decodificación del mensaje recibido */
        sdvd(g, es_ovr_n0, channel_length, encodedr, sdvdout,pru);
        sdvd(g, es_ovr_n0, channel_length, encodedr1, sdvdout1, pru);


         . . .   Conversión de los arreglos a bytes  . . .


        break;
        case SER_ID_SENT:                              /* Después de envío  */
        break;
        case SER_ID_MODEMSTATUS:  /* Después de modif. de estados de línea */
        break;
    }
  }
  irq_SendEOI( iSerIRQ );                          /* final de la interrupción */
}
```

El proceso de decodificación del arreglo *encodedr* por el procedimiento *sdvd,* nos da como resultado un arreglo de 1's y 0's con una longitud de 4 elementos llamado *sdvdout*, estos cuatro elementos corresponden a la mitad de los bits de un byte del mensaje original. Enseguida se decodifica al arreglo *encodedr1* que nos entrega otro arreglo de 1's y 0's de longitud 4 llamado *sdvdout1*, este contiene los restantes bits del byte del mensaje original. Veamos la siguiente grafica

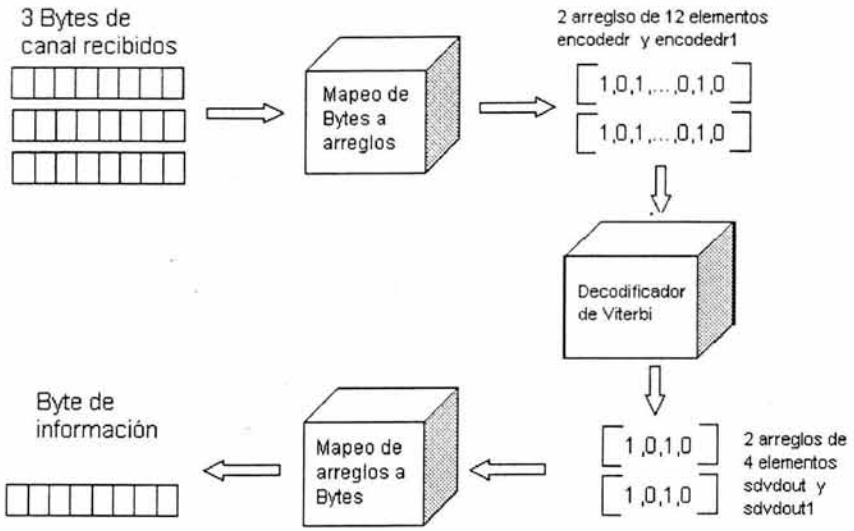


Figura 3.5 Diagrama de proceso decodificación durante la recepción. Observe que de 3 bytes de cabal recibidos se genera un byte de información

Finalmente, ya que tenemos la información necesaria en los arreglos sdvdout y sdvdout1, para reconstruir un byte del mensaje original, tenemos ahora que reunir los dos arreglos en un byte de información. Para ello utilizamos el operador OR que actua bit a bit en los operandos. Siguiendo la misma idea utilizada con el operador AND para poner un bit en particular dentro de un byte de información, lo que hacemos es aplicar el aperador OR entre el byte de información con un byte máscara. El byte máscara es un byte cuyo valor es el peso binario que corresponde al bit que se quiere configurar,es decir, si se quiere configurar el bit menos significativo (el bit más a la derecha), el byte máscara será igual al peso binario $2^0 = 1$, si se quiere configurar el siguiente bit, el byte máscara será igual al peso binario $2^1=2$, asi sucesivamente hasta configurar el bit más significativo, con un byte de máscara igual al peso binario $2^7 = 128$, veamos este último caso gráficamente.

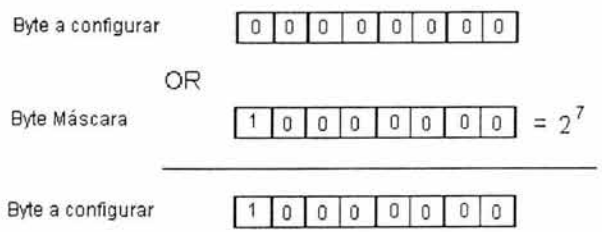


Figura 3.6 Ejemplo de aplicar el operador OR bit a bit entre un byte de canal y un byte "Mascara"

Esta funcionalidad es implementada pasar el contenido de los arreglos al byte de información, simplemente se tiene un arreglo C4 con dos elementos, ambos elementos son puestos a cero, en el primer elemento se van a configurar los bits para que guarde la información del byte recibido, entonces, para los elementos de los arreglos donde su valor es 1, se sabe su peso binario, se calcula dicho peso y se opera el resultado con el elemento de C4 mediante un operador OR, esto configura el bit del elemento C4 que corresponde al peso binario calculado . Después de acabar con todos los elementos de sdvdout y sdvdout1 se tiene en C4 el byte decodificado. Veamos la implementación completa

```
/* GetSer : Rutina de interrupción disparada por el puerto serie              */
/*: En esta función se reciben datos del puerto serie, y se
*/
/*        representan en la ventana *Remote*. Esta función contiene          */
/*        además el esqueleto para todos los demás eventos de puerto   */
/*        serie, que disparen una IRQ.
*/
VOID __interrupt _FP GetSer( VOID )
{ static BYTE bIRQID;
static int i,pot;
static long t;
bIRQID = ( BYTE )inp( iSerPort + SER_IRQ_ID );
if( ! ( bIRQID & SER_ID_PENDING ) )                /* IRQ esperando ? */
{
switch( bIRQID & SER_ID_MASK )
{
case SER_ID_RECEIVED:                          /* Después de recepci¢n */
i = 0;
while( ser_IsDataAvaiable( iSerPort ) && ( i < 16 ) )
{
msgrec[ k++ ] = ( BYTE )inp( iSerPort + SER_RXBUFFER );
msgrec[ k ] = '\0';
}
if (k>2)          /*  Si se han recibido los tres bytes, que corresponden a la
codificación de un solo bit */
{     k=0;
indice=0;
for (t = 0; t < 8 ; t++)      /*guardar en el arreglo el contenido del primer byte
*/
{
pot=(int)pow(2,(7-t));
if ((msgrec[0] & pot)!=0)
*( encodedr + t ) = -1;
else
*( encodedr + t ) = 1;
}
for (t = 0; t < 8 ; t++)     /*guardar en el arreglo el contenido del segundo byte
*/
{
pot=(int)pow(2,(7-t));
if ((msgrec[1] & pot)!=0)
if (t<4)
*( encodedr + (t+8)) = -1;
else
*( encodedr1 + (t-4)) = -1;
else
if (t<4)
*( encodedr + (t+8)) = 1;
else
*( encodedr1 + (t-4)) = 1;
}
for (t = 0; t < 8 ; t++)    /*guardar en el arreglo el contenido del tercer byte  */
{
pot=(int)pow(2,(7-t));
if ((msgrec[2] & pot)!=0)
*( encodedr1 + (t+4)) = -1;
else
```

```
*( encodedr1 + (t+4)) = 1;

}

/* aquí se hace la decodificación del mensaje recibido */
sdvd(g, es_ovr_n0, channel_length, encodedr, sdvdout,pru);
sdvd(g, es_ovr_n0, channel_length, encodedr1, sdvdout1, pru);
c4[0]=0;
c4[1]=0;
/*  El resultado de la decodificación se expresa como un caracter
que se almacena en c4 */
for (t=0; t<4; t++)
{       if (*(sdvdout + t)==1)
{       pot=(int)pow(2,(7-t));
c4[0]=c4[0]|pot;
}
}
for (t=0; t<4; t++)
{       if (*(sdvdout1 + t)==1)
{       pot=(int)pow(2,(7-(t+4)));
c4[0]=c4[0]|pot;
}
}
/* Se muestra el caracter encontrado, en la ventana de mensajes recibidos  */
win_Print( &Remote, c4 );
}
break;
case SER_ID_SENT:                              /* Después de envío  */
break;
case SER_ID_MODEMSTATUS:  /* Después de modif. de estados de línea */
break;
}
}
irq_SendEOI( iSerIRQ );                              /* final de la interrupción */
}
```

3.1.5.4 El envío de la Información

La transmisión de la datos se realiza byte por byte, estos datos pueden ser un carácter introducido desde teclado o un byte que se lee desde archivo. El programa de transmisión realiza el mismo procesamiento de los datos, independientemente de cuál sea su origen. Este proceso que se aplica consiste básicamente de los siguientes pasos:

- Tomar el byte de información. Este byte puede ser el carácter leído desde teclado o puede ser un byte que se obtuvo de la lectura de un archivo.
- Extraer los bits que conforman al byte de información, guardar estos bits en un arreglo, con la finalidad de proveer una mayor facilidad para la manipulación a nivel bits de los datos.
- Hacer una codificación convolucional tomando como entrada a los arreglos que se obtuvieron en el paso anterior. Este tipo de codificación presenta un grado de complejidad considerable en la manipulación de la información a nivel de bits, por lo que se hizo necesaria la utilización de una estructura de almacenamiento temporal de la información a nivel bit, estas estructuras fueron los arreglos.
- Regresar los resultados de la codificación, que esta en un arreglo, al nivel de byte.
- Hacer el envío de la información, ahora codificada, a través del puerto serie.

De esta forma, tenemos que considerar lo siguiente. Tenemos un codificador convolucional para un código de tasa 1/2, K = 3, m = 2. Esto nos trae como consecuencia que la longitud del mensaje después de pasar por la codificación tendrá una longitud de

$$Longitud\_msg\_codificado = (long\_msg + m) * 2$$

Si tratamos de pasar un byte por el proceso de codificación, y dado que contiene 8 bits (*long_msg* = 8) tendríamos como resultado los siguiente:

$$Longitud\_msg\_codificado = (8 + 2) * 2 = 20$$

Es decir, 20 bits que pasaríamos a través del puerto serie y dado que el puerto esta configurado para transmitir palabras de 8 bits, tendríamos que formar dos palabras de 8 bits y completar la ultima con 4 bits cualesquiera, teniendo en consideración, que estos 4 bits de "relleno" tendrían que ser conocidos por el receptor para que éste no los tomará como parte del mensaje original, la siguiente gráfica representa este escenario:

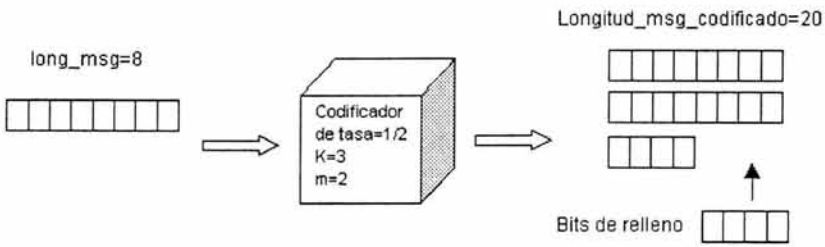


Figura 3.7 Una posibilidad para trasnsmitir un byte (mensaje de 8 bits) es meterlo tal cual en el codificador, se obtiene entonces un mensaje codificado de 20 bits de longitud, por lo que tendríamos que agregar bits de relleno para completar los tres bytes y poderlo transmitir de inmediato.

Sin embargo si tomamos los 8 bits del byte de dato y formamos dos grupos de 4, cada uno de estos dos grupos de 4 bits podría ser codificado de forma independiente y ahora la longitud del mensaje codificado resultante para los dos grupos de 4 bits, sería la misma:

$$Longitud\_msg\_codificado = (4 + 2) * 2 = 12$$

Por lo que si juntamos los mensajes codificados de los dos grupos de 4 bits tendríamos una longitud total igual a la suma de cada una de las longitudes de los dos mensajes, esto es

$$Longitud\_msg\_codificado = 24$$

Entonces, podemos ahora formar 3 palabras de 8 bits sin incluir ningún bit de relleno, por lo que el proceso de codificación se realizaría de forma "natural" y en el lado del receptor no se tendrían que hacer consideraciones especiales, como la de conocer información ajena a la del mensaje original., observe este caso en la ilustración siguiente.

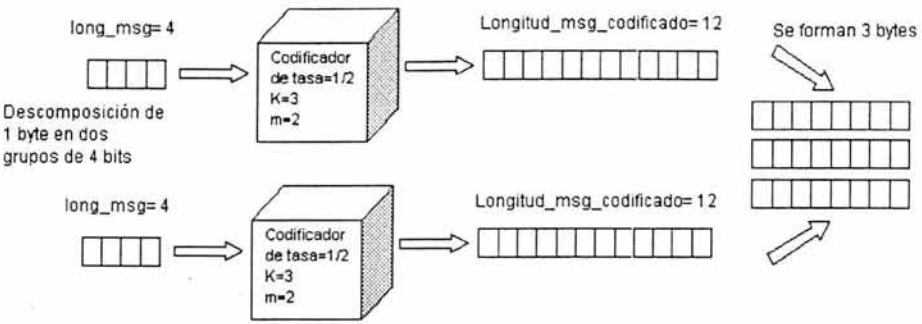


Figura 3.8 El proceso que se elijió para trasnsmitir un byte (mensaje de 8 bits) es descomponerlo en dos mensajes de 4 bits y meterlos en el codificador de forma individual, se obtiene entonces un mensaje codificado de 24 bits de longitud, por lo que tendríamos los tres bytes que podemos transmitir de inmediato.

```
/* Se guardan los bits del caracter leído en dos arreglos, cada uno guarda 4 bits*/
       for (t = 0; t < 8 ; t++)
                        {
          pot=(int)pow(2,(7-t));
          if ((c[0] & pot)!=0)
          /* se guardan los bits 1's */
               if (t<4)
                    /*  Aquí se guarda los 1's de los primeros 4 bits del byte que
se leyó
                        del archivo  */
                  *( onezer + t ) = 1;
               else
               {  /*  Aquí se guarda los 1's de los últimos 4 bits del byte que se
leyó
                        del archivo  */
                  *( mensaje + h ) = 1;
                  h=h+1;
               }
          else
               if (t<4)
                    /*  Aquí se guarda los 0's de los primeros 4 bits del byte que
se leyó
                        del archivo  */
                  *( onezer + t ) = 0;
               else
               {  /*  Aquí se guarda los 0's de los últimos 4 bits del byte que se
leyó
                        del archivo  */
                  *( mensaje + h ) = 0;
                  h=h+1;
               }
        }
       /*   se procede a codificar los primeros cuatro bits del mensaje  */
        cnv_encd(g, msg_length, onezer, encoded);
       /*   se procede a codificar los primeros cuatro bits del mensaje  */
        cnv_encd(g, msg_length, mensaje, encoded1);
       /* Se inicializan los arreglos que van a guardar la información codificada
como
         si fuera un byte */
        c1[0] = 0;
        c1[1] = 0;
        c2[0] = 0;
        c2[1] = 0;
        c3[0] = 0;
        c3[1] = 0;
```

```
        /* Inicia el guardar la información codificada como si fuera byte*/
        /* Para el primer byte a transmitirse toman los primeros 8 bits del arreglo
encoded*/
                    for (t=0; t<8; t++)
         {      if (*(encoded + t)==1)
                {       pot=(int)pow(2,(7-t));
                        c1[0]=c1[0]|pot;
                }
        }
        /* Para el segundo byte a transmitirse se toman los últimos 4 bits del
arreglo encoded
           y los primeros 4 bits del arreglo encoded1*/
         for (t=0; t<4; t++)
            {  if (*(encoded + (t+8))==1)
               {       pot=(int)pow(2,(7-t));
                       c2[0]=c2[0]|pot;
               }
         }
         for (t=4; t<8; t++)
            {  if (*(encoded1 + (t-4))==1)
               {       pot=(int)pow(2,(7-t));
                       c2[0]=c2[0]|pot;
               }
         }
        /* Para el tercer byte a transmitirse toman los últimos 8 bits del arreglo
encoded1*/
         for (t=0; t<8; t++)
            {  if (*(encoded1 + (t+4))==1)
               {       pot=(int)pow(2,(7-t));
                       c3[0]=c3[0]|pot;
               }
         }
         /* pasar caracter entrado por el puerto serie... */
        /* ser_WriteByte( iSerPort, c1[0], 0x8000, 0, 0 );
         ser_WriteByte( iSerPort, c2[0], 0x8000, 0, 0 );
         ser_WriteByte( iSerPort, c3[0], 0x8000, 0, 0 );*/
```

3.2 Los Módulos de transmisión inalámbrica

Una de las principales premisas para la construcción de este proyecto era la utilización de componentes de bajo costo y que estén fácilmente disponibles, de hecho, los módulos utilizados, fueron parte de un requerimiento para la implementación del sistema. Dadas sus características de bajo costo, fácil implementación y a que ya se tenían disponibles estos componentes, se hizo uso de ellos.

3.2.1 Descripción General

Hay tres tipos de módulos inalámbricos de RF (vea apendices) que operan en el rango de 433.92 MHz, estos son: Transmisor, Receptor y un Transceptor(Transceiver). Estos módulos de RF fueron diseñados por la compañía PARALLAX Inc. para servir como herramienta para realizar experimentos relacionados con la comunicación inalámbrica. Los módulos de RF están en una PCB (Printed Circuit Board) con un peine de 17 patillas de 0.1 pulgadas de espaciamiento que cabe directamente en la mayoría de todos los tableros de prototipos, incluyen una antena de loop en la PCB. Son fácilmente integrables en tableros que incluyan codificadores, decodificadores, direccionamiento, proceso de datos con RF e incluso la antena, probada en completo rango, esta lista para su uso. Solamente se

aplica la alimentación de +5 VDC, GROUND y la conexión de las patillas de comunicación que sean requeridas, y el módulo estará listo para establecer comunicaciones inalámbricas.

Existe diferentes formas de utilizar los tres módulos de acuerdo al tipo de comunicación que se quiera implantar, estos son:

- La sola comunicación de una sola dirección requiere por lo menos:
  Opción a: 1 transmisor y 1 receptor.
  Opción b: 1 transmisor y 1 transmisor-receptor.
  Opción c: 1 transmisor-receptor y 1 receptor.

- La comunicación bidireccional requiere por lo menos:
  2 transmisores-receptores.
- Las comunicaciones Multi-punto se pueden alcanzar con:
  Colocando un transmisor en cada nodo que necesita enviar la información.
  Colocando un receptor en cada nodo que necesita recibir la información.
  Colocando un transmisor-receptor en cada nodo que necesita enviar y recibir la información.

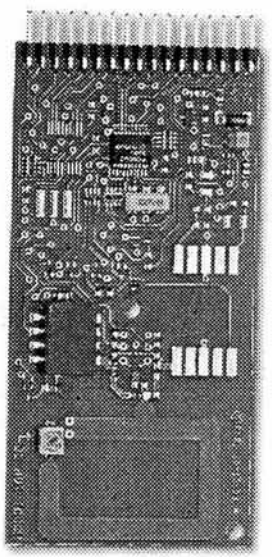


Figura 3.10 Transceiver (transmisor y receptor) de 433.92 MHz , fabricado por PARALLAX Inc.

El transmisor, el receptor y el transmisor-receptor todos tienen interfaces serie independientes de 9600 baudios, 3 terminales de entradas y salidas. Los módulos se

pueden comunicar en distancias de hasta 76 metros. Los circuitos operan con +5V y se interconectan fácilmente a las interfaces de desarrollo Basic Stamp 2 o Basic Stamp 2sx, proporcionadas por la misma empresa En estas interfaces de desarrolla se pueden construir proyectos con un lenguaje de programación para el microprocesador que está en estas tablas de desarrollo, de tal manera que podemos brindar capacidad de manipulación y procesamiento de los datos independiente al tipo de equipo al que se conecte los módulos de RF, para este proyecto se decidió que al procesamiento de la información transmitida se realizaría dentro del equipo terminal que hace uso de la información, en este caso, dentro de la PC. Se tomó esta decisión, dada la capacidad y velocidad de procesamiento, que son mucho mayores en una PC que en la tabla de desarrollo Basic Stamp 2 de Parallax, disponible para nuestro uso.

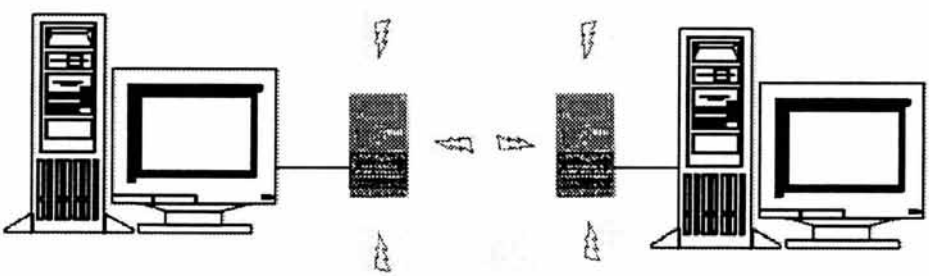


Figura 3.11 El sistema permite la transmisión entre dos o más terminales sin necesidad de cableado, para establecer comunicación serial bidireccional

3.2.2 Dos Modos de operación:

Conectando GND al pin de Modo coloca el módulo en modo difusión.
Conectando +5V al pin de Modo coloca al módulo en modo Serie.

3.2.2.1 Modo difusión:

El transmisor, el receptor y el transmisor-receptor tienen 4 pins de dirección (etiquetados como ADDR1 - ADDR4), proporcionando 16 combinaciones de dirección. Colocando 0V o 5V en los 4 pins de dirección elegimos la dirección de las unidades (en una manera binaria). Por ejemplo, la colocación de 0V en todos los pernos fija la dirección a cero. La colocación de 5V en todos los pernos fija la dirección a 15.
El transmisor, el receptor y el transmisor-receptor también tienen 3 pins de datos (etiquetados como IN1 - IN3). Los niveles lógicos 0V o 5V puestos en los pins de la entrada del módulo que transmite se envían automáticamente a los pins de la salida

(etiquetados como OUT1 - OUT3) en el módulo de la recepción. Además, 16 diversos módulos se pueden direccionar con la incorporación de los pins de dirección de 4-bits . El receptor recibirá los datos sobre sus 3 pins de la salida solamente cuando su dirección de 4-bits corresponda a la dirección del transmisor de 4-bits.

3.2.2.1 Modo serie

En modo serial (con +5V aplicado al pin de Modo) los módulos pueden enviar y recibir datos seriales en 9600, N, 8, 1 con niveles lógicos de +5V y 0V. Conectando simplemente un solo alambre con el pin de transmisión de datos (etiquetado con TXD) y se envían los datos a 9600 baudios en el módulo. El módulo de la recepción hace salir los mismos datos en 9600 baudios. Toda el procesamiento de datos en RF es hecha automáticamente por los módulos. Un pin de control de flujo es proporciona en el lado que transmite para ayudar a alcanzar máximo rendimiento de procesamiento. El Parallax Basic Stamp 2 y el BASIC Stamp 2sx tienen comandos incorporados para establece la transmisión de bytes seriales y control de flujo en una sola instrucción.

Para la implementación del sistema se utilizó el modo serie dada la compatibilidad con la transmisión serial tipo de puerto utilizado.

3.2.3 RS 232

Dado que el puerto serie esta descrito por el estándar RS-232, mencionaremos que la EIA fue una de las organizaciones que elaboró este estandar, sin embargo, la EIA no es la única institución de normalización que aprobó el estándar RS-232. También el internacional CCITT (Comitè Consultatif International Télégraphic et Téléfonique) aceptó esta norma, pero la dividió en dos: V.24 y V.28. En la norma V.24 se especifican los protocolos de comunicación asíncronos y el modo de conexionado de los conectores, así como su formato. En la norma V.28 se incluyó el resto de la especificación RS-232, los valores eléctricos característicos. De entre ellos podemos citar el nivel de señal, la impedancia terminal y la resistencia a cortocircuito. *El nivel de señal debe moverse entre +3 voltios y +15 voltios para el cero lógico y entre -3 voltios y -15 voltios para el uno lógico.*

Dado que el estándar RS-232 puede extenderse a cualquier sistema, no se habla nunca en la especificación de un tipo de prodesador concreto, sino siempre de los dos polos, el Data Terminal Equipment (DTE) y Data Communication Equipment (DCE). En el caso concreto de las transmisiones vía PC y módem, el PC sería el DTE mientras que el módem representaría el DCE. Ambos están unidos mediante un cable RS-232, obviamente el cable deja de ser necesario cuando se tiene instalada una tarjeta inalámbrica en la PC. En este caso, la tarjeta constituye por sí misma el puerto serie.

Para la transmisión serie en un sentido se precisan únicamente dos líneas (GROUND y datos) y para la transmisión en los dos sentidos, sólo tres (una más para la línea de datos adicional en el sentido contrario). Además, tanto EIA como CCITT han descrito e introducido diecisiete líneas más cuya función es exclusivamente de control y gestión. De todos modos, y afortunadamente, no se tienen que implementar todas las líneas

para llevar a cabo una transmisión RS-232. De lo contrario, una de las ventajas más importantes frente a los cables paralelo se habría perdido.

Lo que aumenta de esta manera el conector son, entre otras, seis líneas adicionales a través de las cuales se entienden el DTE (PC) y el DCE (módem). RTS, CTS, DSR y DTR sirven básicamente para regular la intercomunicación entre el PC y un módem cuando ésta se produce en ambos sentidos (full duplex mode), caso más frecuente.. Vea la siguiente tabla

Linea	Abreviatura	Significado
Transmitted Data	TxD	Los datos se transmiten a través de esta línea. Sin embargo el DTE (PC) sólo puede empezar a enviar cuando las cuatro líneas de gobierno RTS, CTS, DSR y DTR tengan un uno lógico. Según el protocolo V.24 esta línea se encuentra en marking condition (lógico 1) cuando no se transmiten datos.
Received Data	RxD	La línea de datos del DCE (modem) al DTE (PC).
Request To Send	RTS	Poniendo esta línea a uno 1 lógico, el DTE (PC) pregunta al DCE (modem) si está preparado para recibir datos.
Clear To Send	CTS	Tras un RTS, el DCE (modem) pone esta línea en 1 lógico, tan pronto como está preparado para recibir datos.
Data Set Ready	DSR	Poniendo esta línea a 1 lógico, el DCE (modem) indica al DTE (PC) que se ha establecerse contacto con la parte contraria (se ha marcado exitosamente el número correspondiente) y pueden por tanto enviarse datos al lejano DCE (otro modem).
Date Terminal	DTR	El DTE (PC) pone esta línea en 1 lógico tan pronto como
Ready		está preparado para comunicarse con el DCE (modem). Con ello el modem sabe que está conectado a un DTE activo.
Ring Indicator	RI	A través de esta línea el DCE (modem) indica al DTE (PC) de que hay una llamada en la línea telefónica a la que está conectada el modem.
Received Line	RLSD	A través de la línea RLSD el DCE (modem) indica al DTE Signal Detector(PC) que ha recibido una señal (carrier) desde el otro extremo de la línea. Por eso se habla también de carrier detect. Esto no implica sin embargo que ya se haya establecido un verdadero contacto, pues bajo ciertas circunstancias puede darse que ambos DCE (módems) no encuentren un protocolo de transmisión común en cuanto a modulación/demodulación.

Tabla 3.7 Significado de las líneas de gobierno

Otro punto importante a considerar es que: **Los pins de TXD y de RXD son de lógica de +5V. NO SUPORTAN NIVELES VOLTAICOS RS232, no se debe conectar directamente con el puerto RS232.** Esto es, los pins seriales de entrada y de salida funcionan en los niveles lógicos +5V y 0V, no se pueden conectar directamente con un puerto de la computadora RS-232, pues esto daña al módulo. Los niveles típicos en un puerto de la computadora RS232 son +10V y -10V; estos voltajes dañarían inmediatamente el módulo. El módulo se ha diseñado para interconectar directamente con la Basic Stamp 2, la BASIC Stamp 2sx que son tablas de desarrollo proporcionadas por la misma compañía, y otros dispositivos de lógica de +5V.

Por lo tanto dado que los pins de TXD(transmisión) y de RXD (Recepción) son los que vamos a utilizar no se hizo la conexión directa con el puerto RS232., se tuvo que implementar un circuito para acoplar la lógica de +5V con la lógica del RS232. La idea del circuito acoplador es :

- Antes de pasar o recibir señales del módulo inalámbrico tenemos que hacer la adecuación de los voltajes a través de un convertidor de Voltajes TTL a RS232 y viceversa, esto se logró con el uso de un manejador de línea implementado con el Circuito Integrado SP232A. Un pequeño inconveniente, es que también aplica inversión, por lo tanto debemos de aplicar el siguiente paso
- Invertir la lógica para ambos puertos: el puerto de entrada para el módulo (TXD) y el de la salida(RXD) . Esto se logra mediante el uso de un inversor de la familia TTL, para nuestra implementación se utilizó el Circuito Integrado SN74LS04.

Una vez hecho este acoplamiento, la comunicación se realiza en forma directa entre el puerto serie y el módulo inalámbrico.

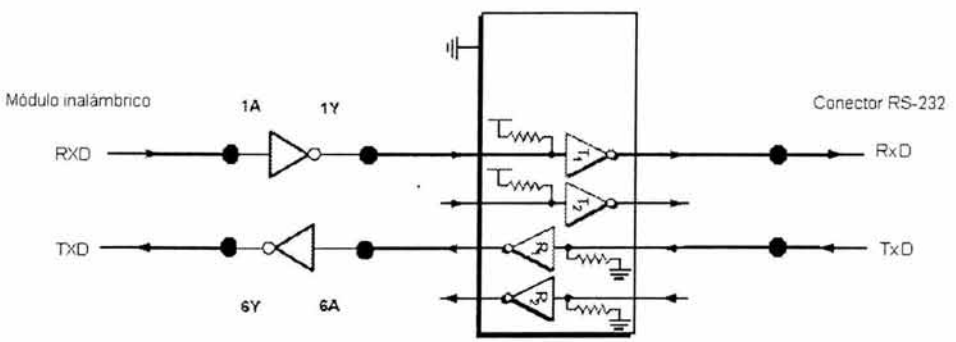


Figura 3.12 Diagrama de Lógico para hacer compatibles los niveles RS-232 del puerto serie y la lógica de 5V de las tarjetas de transmisión

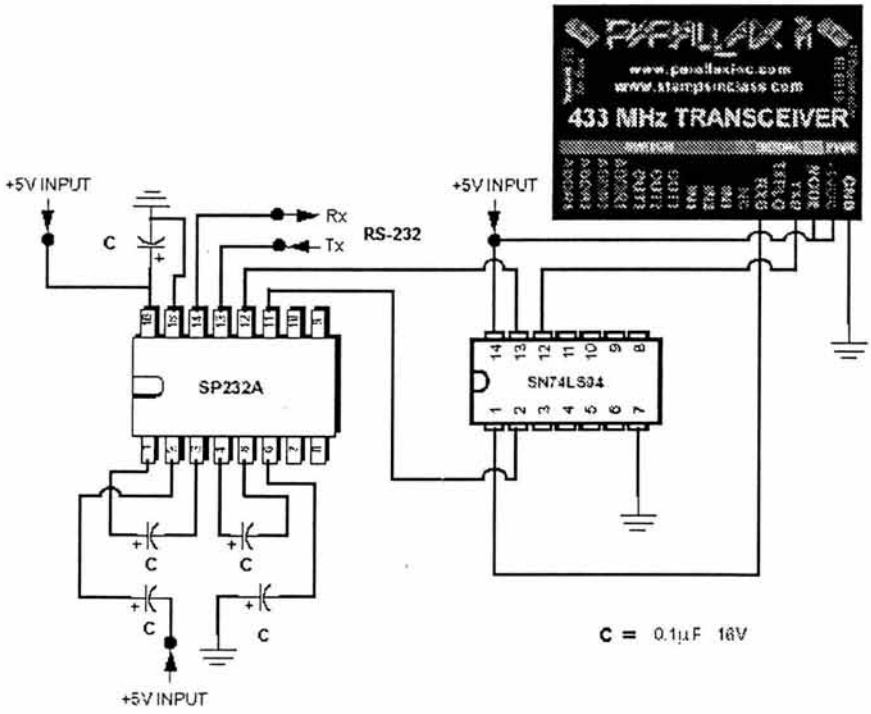


Figura 3.13 Diagrama de conexiones para hacer compatibles los niveles RS-232 del puerto serie y la lógica de 5V de las tarjetas de transmisión

Capítulo 4

Uso de la Interfaz de Usuario

Capítulo 4. Uso de la interfaz de usuario

4.1 Iniciando la aplicación

La interfaz de usuario es una aplicación tipo texto que es operada en línea de comando. Para comenzar la aplicación, debemos llamar al archivo ejecutable "SERTRANS.EXE". Veamos las siguientes gráficas que ilustran su funcionamiento. Primero tenemos una gráfica que muestra la ayuda en línea de comando. Al poner el signo de interrogación como parámetro en el llamado de la aplicación, se despliega una lista de los parámetros que lleva el comando, estos parámetros son para la configuración del puerto y velocidad de trasmisión. Posteriormente, ilustramos un llamado típico con el uso de los dos parámetros, el parámetro del número de puerto serie y el de velocidad de transmisión. Además se muestra la pantalla de entrada de la aplicación.

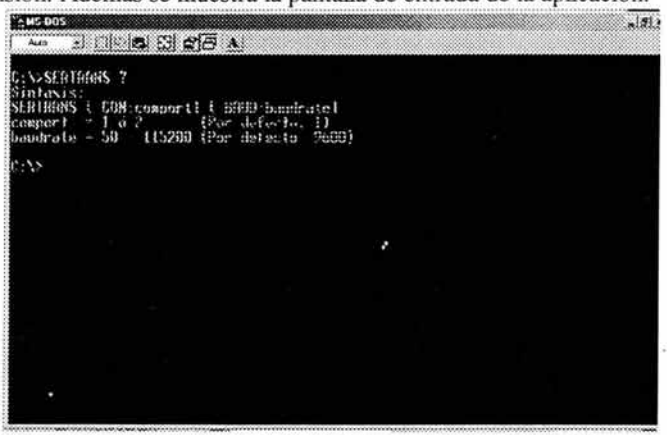


Figura 4.1: Ejecución del programa mostrando el uso del parámetro de ayuda

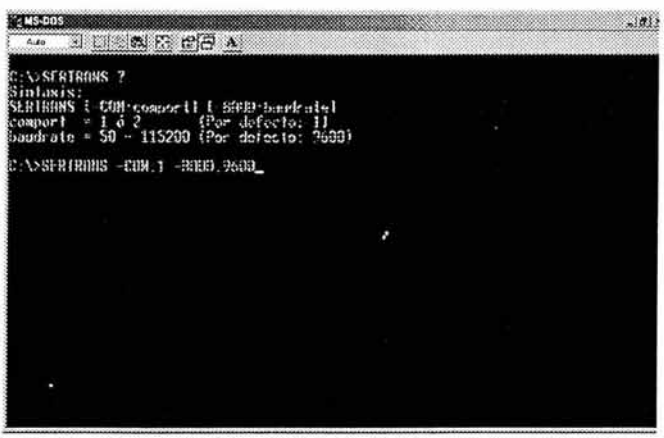


Figura 4.2 Ejemplo de llamado del programa para usar el puerto serie 1 y una velocidad de trans misión de 9600 bps

4.2 Características de la aplicación

La aplicación esta dividida en tres zonas:

- En la zona superior de la pantalla, llamada zona *"REMOTA"*, se despliegan los mensajes que fueron transmitidos desde algún otro nodo de comunicación y son recibidos en nuestra estación de trabajo.
- En la zona inmediata inferior de la pantalla, llamada zona *"LOCAL"* se muestran los mensajes que enviamos desde nuestra estación y también es la zona de "COMANDOS" para realizar alguna otra operación.
- Por último en la parte inferior de la pantalla se encuentra la zona en donde se despliega los parámetros de configuración del puerto serie usados en el establecimiento de la comunicación

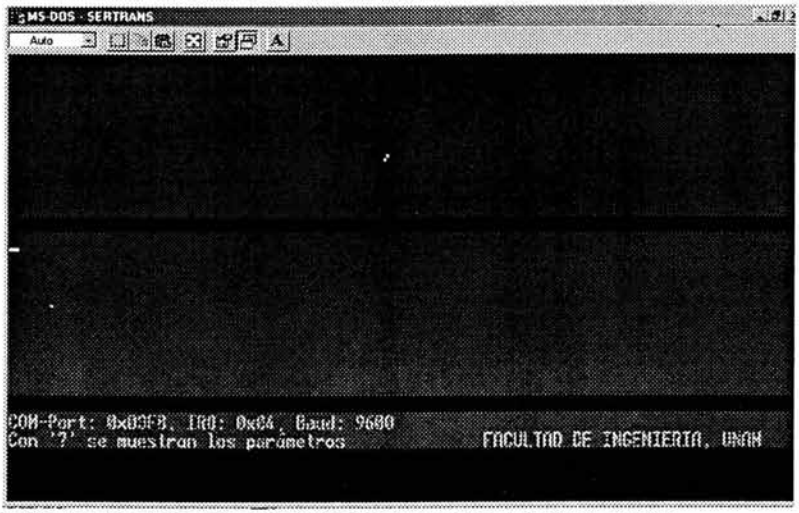


Figura 4.3 Pantalla de presentación para el llamado de programa de la figura anterior

El siguiente par de graficas ilustra un tipo de comunicación que se puede establecer con esta interfaz. Un usuario en una estación, puede escribir cualquier mensaje en la zona "LOCAL" de su interfaz, este mensaje se codifica y se transmite inmediatamente carácter por carácter a través del puerto serie, mientras que otro usuario en otra terminal recibe el mensaje tal y como se va tecleando en la parte transmisora y se visualiza en la zona "REMOTA" de su interfaz.

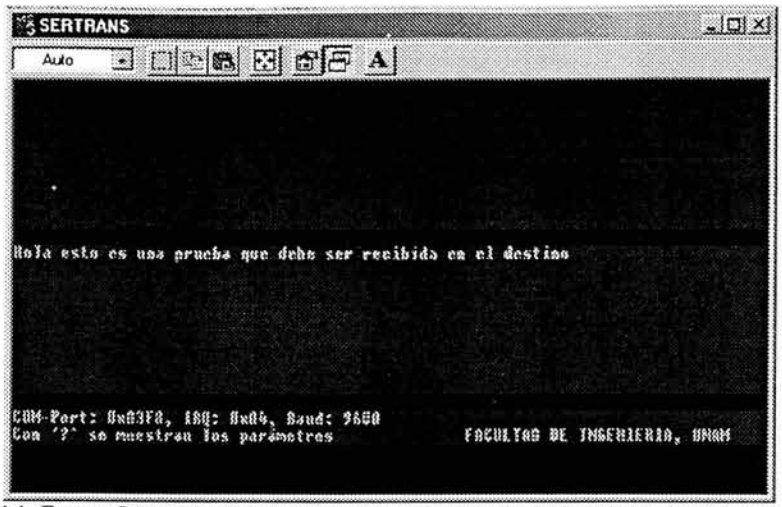


Figura 4.4 En esta figura se muestra un mensaje que fue entrado desde teclado, el mensaje se codifica y envía caracter por caracter y se despliega en la zona "LOCAL" de la aplicación

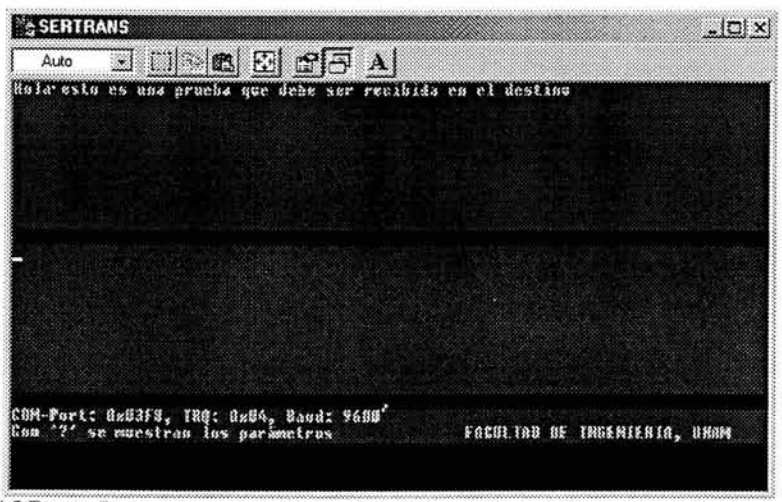


Figura 4.5 En esta figura se muestra el mensaje enviado visto en la estación receptora, el mensaje se recibe y se despliega en la zona "REMOTA" de la aplicación. El mensaje se va desplegando tal y como se va introduciendo del lado estación transmisora.

Al entrar a la aplicación nos situamos en modo "TEXTO", donde cualquier carácter que escribamos es inmediatamente codificado y transmitido.Sin embargo podemos entrar a otro modo, llamado modo comando, en este modo podemos ejecutar comandos básicos para el manejo de la aplicación. Para entrar en modo comando debemos presionar la tecla <ESC>, y a continuación aparecerá el mensaje "Entre el comando:", en este momento la aplicación espera que se proporcione un comando valido para ejecutarlo. A continuación se muestra una tabla con los comandos que se implementaron en esta aplicación.

Comando	Nombre	Descripción
e	Enviar	Enviar archivos texto
r	Recibir	Pone a la aplicación de tal manera que todo lo que se reciba se guarde en un archivo
:	Modo Texto	Pone a la aplicación en modo texto
x	Salir	Salir de la aplicación

Al entrar en modo "COMANDO", la zona "LOCAL" se convierte a zona de "COMANDOS" y se deje de transmitir la entrada que se recibe del teclado. Vea la siguiente gráfica.

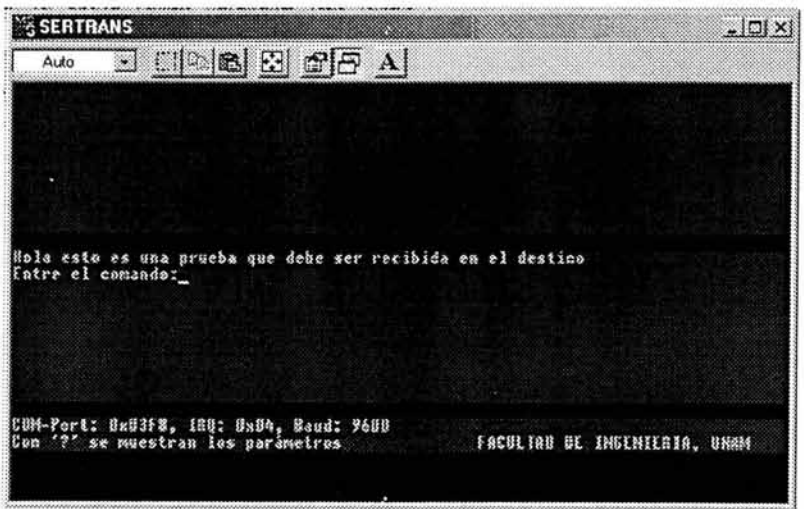


Figura 4.6 Se muestra el estado "COMANDO" en que entra la aplicación al presionar la tecla <ESC>

4.3 Enviando un archivo

Si se desea enviar un archivo, se debe hacer lo siguiente:

- Se debe entrar en modo "COMANDO" con la tecla <ESC>
- Entrar el carácter "e " que corresponde al comando de envío.
- Una vez entrado este comando se solicita el nombre del archivo a enviar, el cual debe ser proporcionado. Una vez entrado el nombre, se presiona la tecla <ENTER>
- La aplicación responde con el nombre del archivo entrado y comienza a enviar.

Cada carácter del archivo es codificado y enviado a través del puerto serie, una vez terminada la transmisión del archivo, la aplicación notifica la finalización de la transmisión y regresa a modo "COMANDO". Veamos la secuencia de este procedimiento.

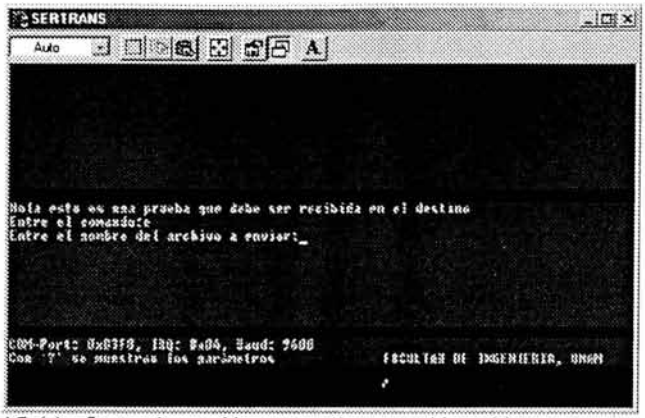


Figura 4.7 (a) Secuencia seguida para enviar un archivo (a) se entra en modo "COMANDO" y se proporciona el comando "e"

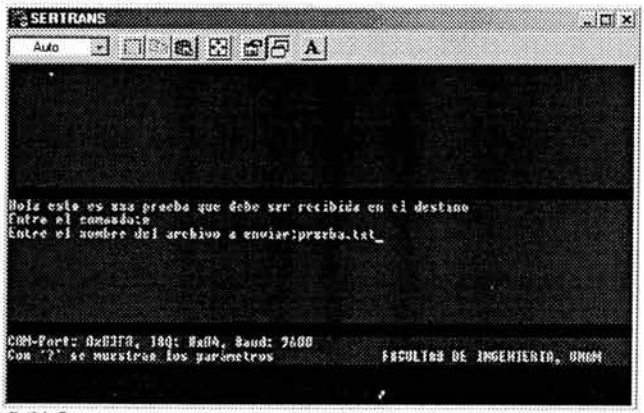


Figura 4.7 (b) Se proporciona el nombre del archivo a enviar

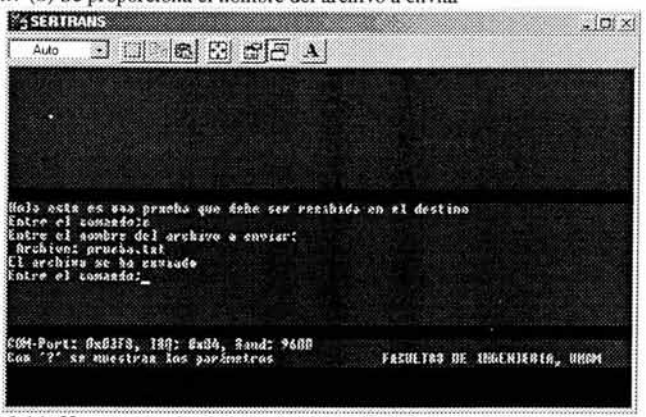


Figura 4.9 (c) Una vez terminada la transmisión del archivo se muestra el mensaje de enviado y se regresa al modo comando.

4.4 Recibiendo un archivo

Mientras tanto del lado de la estación receptora podemos hacer que la aplicación guarde toda la información, que se esté recibiendo en el puerto serie, en un archivo que le especifiquemos. Para realizar esta tarea, debemos hacer lo siguiente:

- Poner a la aplicación en modo "COMANDO" con la tecla <ESC>
- Proporcionar el comando "r". La aplicación solicita el nombre del archivo, es entonces cuando se introduce el nombre del archivo, se presiona la tecla <ENTER>
- La aplicación responde con dos mensajes para avisar que se ha comenzado a guardar en un archivo lo que se está recibiendo. Un mensaje se despliega en la zona "LOCAL" de la pantalla, y consiste en la frase "Recibiendo archivo (Presione f para finalizar): ", seguido del nombre del archivo que se proporcionó. El segundo mensaje se despliega en la zona "REMOTA" y consiste del mensaje "Se está guardando: ", este mensaje es para indicar a partir de dónde, dentro de todo lo que se esta recibiendo, se comenzó a guardar.
- Presionar "f " para finalizar la recepción

Veamos las siguientes gráficas que ilustran el proceso descrito:

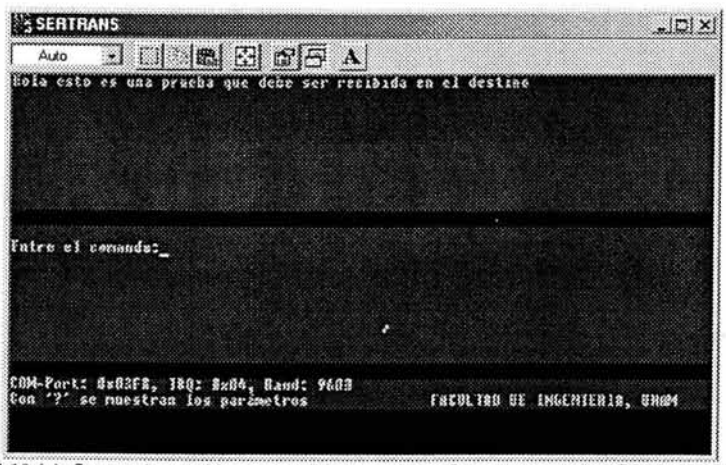


Figura 4.10 (a) Secuencia seguida para recibir un archivo. Se entra en modo "COMANDO"

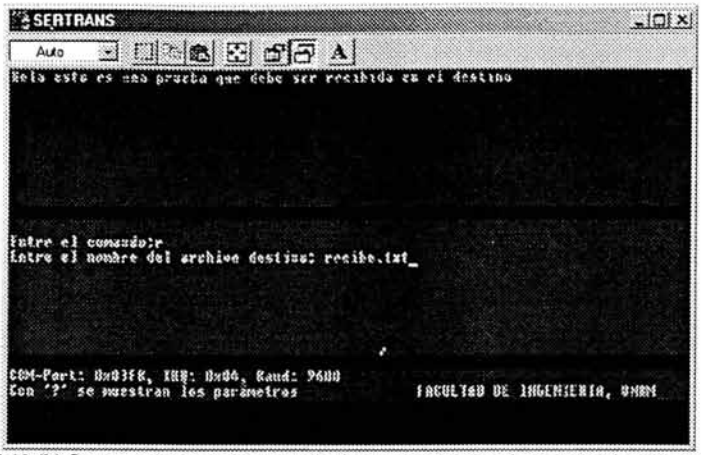


Figura 4.10 (b) Se proporciona el comando " r " y se proporciona el nombre del archivo a enviar

Figura 4.10 (c) La aplicación manda un mensaje en la zona "LOCAL" para avisar que esta listo para recibir

Figura 4.10 (d) Se manda otro mensaje de aviso en la zona "REMOTA" para indicar que lo que se reciba a partir de ese momento se esta guardando en archivo

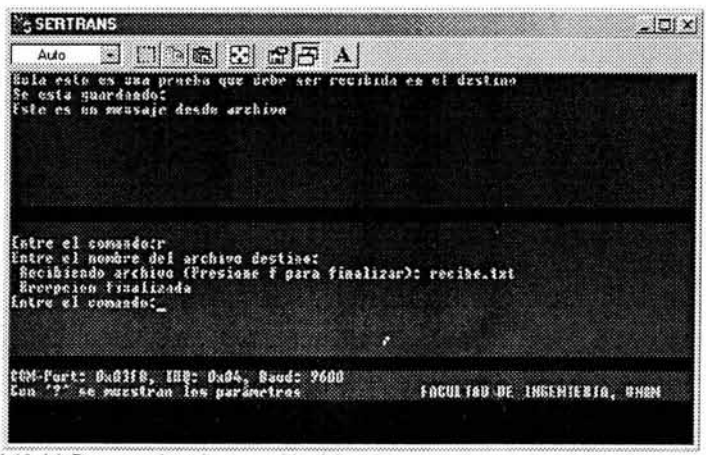


Figura 4.10 (e) Para terminar la recepción del archivo se muestra el mensaje de finalizado y se regresa al modo comando.

Una vez finalizada la recepción se regresa a modo "COMANDO".

Para finalizar la aplicación se entra en modo "COMANDO" con la tecla <ESC> y se presiona la tecla "x", que corresponde al comando salir.

También si se está en modo comando podemos regresar al modo "TEXTO" simplemente proporcionando el caracter ": " que corresponde al comando poner en modo "TEXTO"

Capítulo 5

Resultados

Capítulo 5. Resultados

5.1 Simulación sin Codificación Convolucional

Suponga que tenemos un sistema donde un bit de canal '1' es transmitido como un voltaje de –1 V, y un bit de canal '0' es transmitido como un voltaje de +1 V. Esto es llamado señalización No Retorno a Cero bipolar (bipolar NRZ). El receptor esta comprendido por un comparador que decide si el bit de canal recibido es un '1' si su voltaje es menos que 0V, y un '0' si su voltaje es más grande o igual que 0V. Uno quisiera muestrear la salida del comparador en la mitad de cada intervalo de bit de dato.

Veamos ahora como se desempeñó nuestro sistema, primero, cuando la E_b/N_0 es alta, y luego cuando la E_b/N_0 es baja.

La siguiente figura muestra los resultados de una simulación de canal donde un millón (1 x 10^6) de bits de canal son transmitidos a través de un canal AWGN con un nivel de E_b/N_0 de 20 dB (esto es, la señal de voltaje es diez veces el voltaje rms de ruido). En esta simulación, un bit de canal '1' es transmitido en un nivel de –1V, y un bit de canal '0' es transmitido en un nivel de +1V. El eje x de la figura corresponde a los voltajes de señal recibidos, y el eje "y " representa el número de veces que cada voltaje fue recibido. Nuestro receptor simple detecta un bit de canal recibido como ' 1 ' si su voltaje es menos que 0V, y como ' 0 ' si su voltaje es mayor o igual a 0V. Tal receptor tendría poca dificultad el recibir correctamente una señal según lo representa la figura de arriba. Muy pocos (si no es que ninguno) errores de recepción de bits de canal ocurrirán. En esta simulación del ejemplo con la E_b/N_0 en 20 dB, un '0' transmitido nunca fue recibido como ' 1 ', y un ' 1 ' transmitido nunca fue recibido como ' 0 '. Hasta ahora, muy bien.

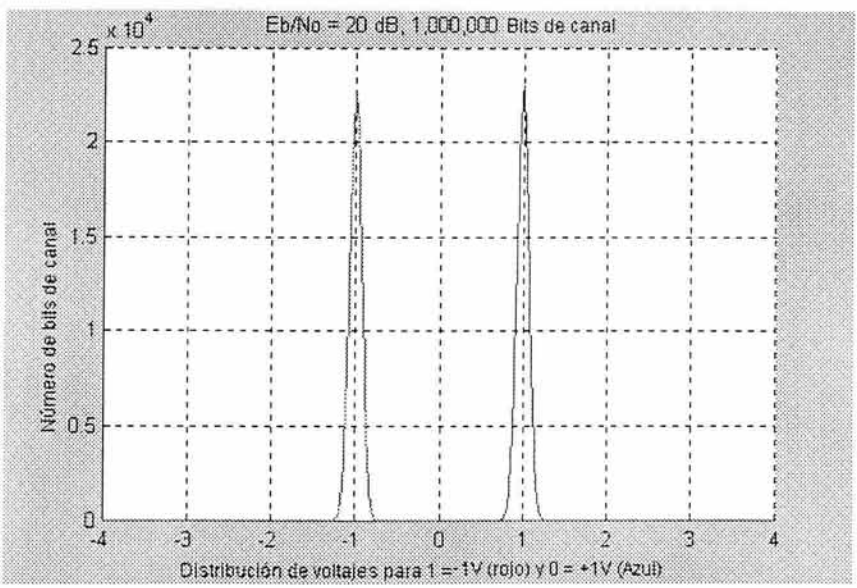


Figura 5.1 Resultados de la simulación de canal donde un millón (1 x 10^6) de bits de canal son transmitidos a través de un canal AWGN con un nivel de E_b/N_0 de 20 dB

La figura siguiente demuestra los resultados de una simulación similar de canal cuando $1x10^6$ bits de canal se transmiten a través de un canal AWGN donde el nivel de E_b/N_0 ha disminuido a 6 dB (es decir el voltaje de la señal es dos veces el voltaje rms del ruido):

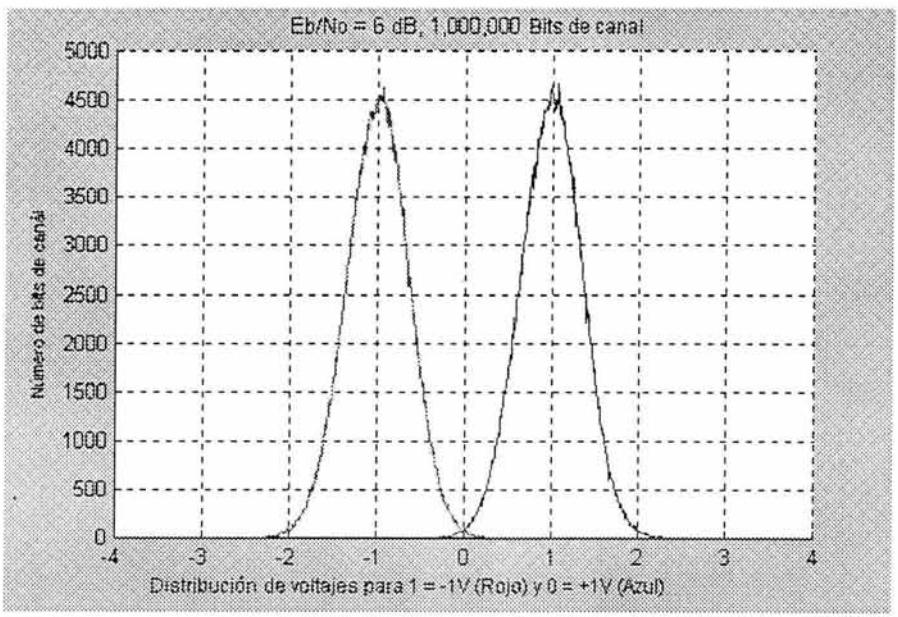


Figura 5.1 Resultados de la simulación de canal donde un millón (1 x 10^6) de bits de canal son transmitidos a través de un canal AWGN con un nivel de E_b/N_0 de 20 dB

Ahora observe cómo el lado derecho de la curva roja en la figura sobrepasa el valor 0V, y cómo el lado izquierdo de la curva azul también cruza 0V. Los puntos en la curva roja que están sobre 0V representan los eventos donde un bit de canal que fue transmitido como uno (-1V) fue recibido como cero. Los puntos en la curva azul que están antes de 0V representan los acontecimientos donde un bit de canal que fue transmitido como un cero (+1V) fue recibido como uno. Obviamente, estos acontecimientos corresponden a los errores de la recepción de bits de canal en nuestro receptor simple. En esta simulación del ejemplo con el sistema de E_b/N_0 en 6 dB, un ' 0 ' transmitido fue recibido como ' 1 ' 1147 veces, y un '1 ' transmitido fue recibido como ' 0 ' 1207 veces, correspondiendo a una tasa de error de bit (BER) de cerca de 0.235%. Eso no es tan bueno, especialmente si se está intentando transmitir datos altamente comprimidos, tales como televisión digital. Se

observará que usando la codificación convolucional con decodificación de Viterbi, se puedo alcanzar una BER mejor que 1 x 10 -7 con la misma E_b/N_0 de 6 dB.

5.2 Simulación con Codificación Convolucional

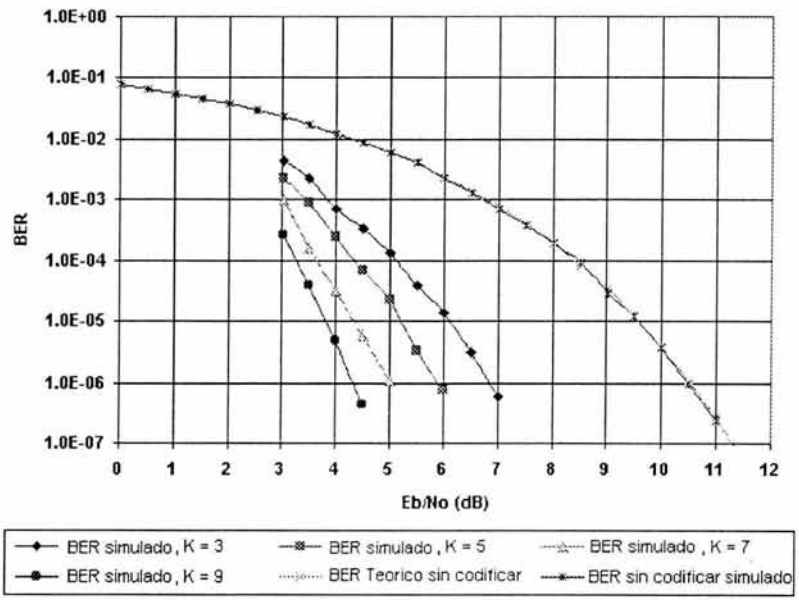


Figura 5.1 Resultados de la simulación para la codificación convolucional con decodificación de Viterbi sobre un canal con AWGN con varias longitudes restringidas

Se obtuvieron los resultados mostrados en esta gráfica usando la simulación con el código, con la profundidad de enrejado fijada a K x 5, usando el cuantizador adaptante con símbolos de canal de tres bits de cuantización. Se corrió la simulación hasta 100 errores (o posiblemente más) ocurridos. Con este número de errores, se tiene una confianza del 95% de que el número verdadero de errores para el número de bits de datos con la simulación esté entre 80 y 120.

Observe cómo los resultados de la simulación para una BER en un canal sin codificar está cercano a la BER teórica para un canal sin codificar, que es dado por la ecuación

$$P(e) = 0.5 * \text{erfc}(\text{sqrt}(E_b/N_0)) = Q(\text{sqrt}(\grave{E}_b/N_0)$$

Sobre el intervalo de confianza del 95%, podemos decir algo. Los acontecimientos del error ocurren como proceso de Poisson, una secuencia de eventos al azar en tiempo. El

proceso de Poisson tiene una tasa media λ igual a n/t, donde n es el número de acontecimientos (el número de errores, en este caso) y t es el intervalo del tiempo de la medida. Para los propósitos de la simulación, vamos dejar t = el número total de bits en la simulación. Vamos a decir que medimos 100 errores en 100 000 bits. La tarifa λ es así 100/100 000, o 1×10^{-3}. Si ajustamos la simulación para correr con 100 000 bits, entonces la media μ de la distribución de Poisson es λ t, o 100 errores. La fórmula para la probabilidad de un número esperado r de errores, dada una media μ de errores, es $P(r) = \frac{\mu^r}{r!} e^{-\mu}$. Tal que la distribución de Poisson para 50 a 150 errores, dada un media de 100 errores, se ilustra en la figura de abajo:

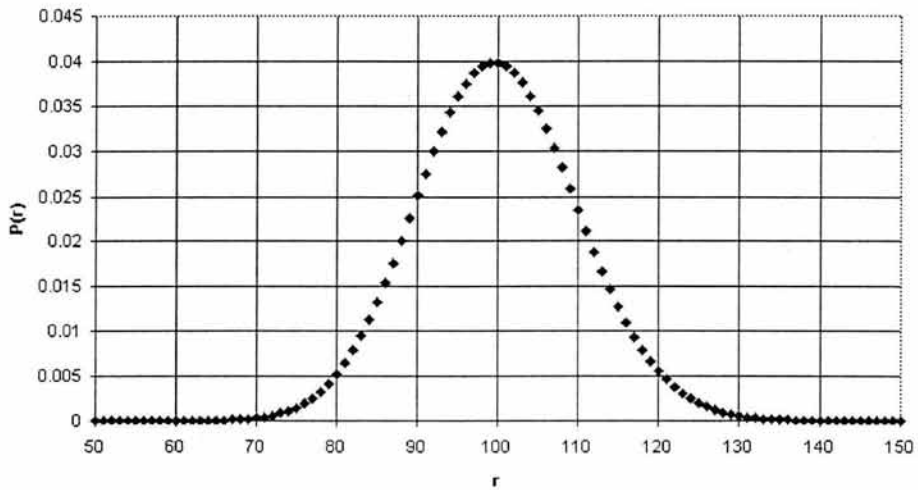


Figura Probabilidad de Poisson P(r) para $\mu = 100$, r = 50 a 150

La probabilidad acumulada de la distribución anterior para un rango de 80 a 120 errores es realmente 95,99% (aproximadamente).

Bibliografía y Referencias

BIBLIOGRAFÍA Y REFERENCIAS

[1] S. Haykin. *Communication Systems*, 3rd ed. New York: John Wiley & Sons, 1994

[2] Leon W. Couch II. *Digital and Analog Communication Systems*, 4th ed. New York: Macmillan Publishing Company, 1993.

[3] A. J. Viterbi. "*Error bounds for convolutional codes as asymptotically optimum decoding algorithm*", publicado en IEEE Transactions Information Theory, vol.IT-13,pp.260-269, abril 1967

[4] José A. Carballar. *El libro de las comunicaciones del PC técnica, programación y aplicaciones*;. ED. Alfa Omega.

[5] Michael Tisher , Bruno Jennrich. PC Interno 5: programación de sistemas. Barcelona. Ed. Marcombo, 1996.

[6] http://home.netcom.com/~chip.f/viterbi/tutorial.htm

[7] http://citel.upc.es/users/buran/buran8/penin/penin.htm

[8] http://trabajospracticos.4mg.com/cod/secc1.htm

[9] http://ceres.ugr.es/~alumnos/c_avila/gsm0.htm

[10]http://com.uvigo.es/asignaturas/scvs/trabajos/curso0001/simula/Simulacion_Bloques/memoria.htm

Apéndice

RF MODULE MANUAL

433.92 MHz TRANSMITTER	433.92 MHz TRANSCEIVER	433.92 MHz RECEIVER
Part # 27986	Part # 27988	Part # 27987
Single Direction Send Only	Bi-Directional Send and Receive	Single Direction Receive Only
Compatible with:	**Compatible with:**	**Compatible with:**
Receiver Part # 27987 Transceiver Part # 27988	Transmitter Part # 27986 Receiver Part # 27987 Transceiver Part # 27988	Transmitter Part # 27986 Transceiver Part # 27988

GENERAL DESCRIPTION

There are three Wireless RF Modules, Transmitter, Receiver and a Transceiver. These RF Modules are designed to serve as a tool for electronic design engineers, developers, hobbyists and students to perform wireless experiments. These modules make it easy for any NON RF Experienced developer to add Wireless RF Remote Control to their project. NO RF Knowledge required. The RF Modules are in a PCB (Printed Circuit Board) form with a 17 Pin 0.1 Inch spacing header that fits directly into most all prototyping boards. They are easy to use boards that include encoders, decoders, addressing, RF data processing and even the antenna, in a simple fully range tested board, that is ready to plug right into your project. Just apply +5VDC, ground, and the communication pins you require and enjoy hassle free wireless communications.

The Transmitter, Receiver and Transceiver all have 9600 baud serial interfaces and stand-alone, 3 function switch inputs and outputs. The modules can communicate over distances up to 250 feet. The boards operate on +5V and easily interface to your Basic Stamp 2 or Basic Stamp 2sx.

Web: http://www.parallaxinc.com or http://www.stampsinclass.com • Email: stamptech@parallaxinc.com
Sales / Technical Support - Tel: (916) 624-8333 Fax: (916) 624-8003
599 Menlo Drive • # 100 • Rocklin, CA 95765

Two Modes of Operation:
Connecting GND to the Mode pin places the module in Switch Mode.
Connecting +5V to the Mode pin places the module in Serial Mode.

Switch Mode:
The transmitter, receiver and transceiver have 4 address pins (labeled ADDR1 – ADDR4), providing 16 address combinations. Placing 0V or 5V on the 4 address pins sets the unit's address (in a binary fashion). For example, placing 0V on all pins sets the address to zero. Placing 5V on all pins sets the address to 15.

The transmitter, receiver and transceiver also have 3 switch data pins (labeled IN1 – IN3). 0V or 5V logic levels placed on the input pins of the transmitting module are automatically sent to the output pins (labeled OUT1 – OUT3) on the receive module. In addition, 16 different modules can be addressed with the built-in 4-bit address pins. The receiver will receive the switch data on its 3 switch output pins only when its 4-bit address matches the transmitter's 4-bit address. The 4-bit address does not apply to serial mode.

Serial Mode
In serial mode (with +5V applied to the Mode pin) the modules can send and receive serial data at 9600, N, 8, 1 with +5V and 0V logic levels. Simply connect a single wire to the Transmit Data pin (labeled TXD) and send 9600 baud data into the module. The receive module outputs the same data at 9600 baud. All RF data processing is done automatically by the modules. You cannot send a continuous 9600 baud stream; spacing between 9600 baud bytes has to be at least 15 milliseconds. A flow control pin is provided for the transmitting side to assist with achieving maximum efficient throughput. The Parallax Basic Stamp 2 and BASIC Stamp 2sx have built-in commands to do serial byte pacing and flow-control handshaking in one single instruction (see source code examples).

MINIMAL REQUIREMENTS

Single direction communication requires at least:
Option a: 1 Transmitter and 1 Receiver.
Option b: 1 Transmitter and 1 Transceiver.
Option c: 1 Transceiver and 1 Receiver.

Bi-directional communication requires at least: 2 Transceivers.

Multi Point Communications can be achieved by:
Placing one transmitter at each node that needs to send information.
Placing one receiver at each node that needs to receive information.
Placing one transceiver at each node that needs to send and receive information.

REGULATORY WARNINGS

- These modules (boards) are not FCC approved. They are designed to comply with the FCC Part 15 Rules and Regulations. They are not in a finished product form. They are strictly intended for experimental purposes only. If you wish to use these modules in an actual product (a non-experimental capacity), the module must first be designed into the product, then the whole product must be approved by the FCC. For a list of FCC approved Labs that can test your final product for compliance contact RF Digital Corporation at (818) 500-1082 or visit their web site at http://www.rfdigital.com.

- It is the responsibility of the user to be aware of the regulatory requirements in their area of operation and application. For exact information contact the FCC Office at:

  Federal Communications Commission
  445 12th St. SW
  Washington DC 20554
  (202) 418-0190
  http://www.fcc.gov

- USE OUTSIDE OF THE U.S.A. It is the responsibility of the user to be aware of the regulatory requirements in their area of operation and application. Contact your local regulatory agency and obtain compliance information.

- For O.E.M. Design-In Guidance for the modules please contact:
  RF Digital Corporation
  1160 North Central Ave. Suite 201
  Glendale, CA 91202
  Tel: (818) 500-1082
  Fax: (818) 246-9122
  Web: www.rfdigital.com
  Email: oem@rfdigital.com or info@rfdigital.com or support@rfdigital.com

OPERATION WARNINGS

- Do not expose the boards to direct outdoor environment. If they will be used outdoors, keep them away from water, moisture and direct sunlight.

- The serial input and output pins operate at +5V and 0V logic levels. Do not attempt to connect directly to a computer RS232 port as this will damage the module. Typical levels at a computer RS232 port are +10V and -10V; these voltages would immediately damage the module. The module is intended to interface directly with the Basic Stamp 2, the BASIC Stamp 2sx and other +5V logic devices.

- Keep the top part of the board (PCB Trace Antenna, opposite end of the 17 pin header) at least 4 inches away from any object, especially metal, wires and batteries as this will de-tune the antenna, drastically reducing performance. Laying the module flat on a work desk made of wood, plastic or glass can sometimes cause problems as well, since some paints contain conductive materials.

- Do not touch the adjustable capacitor located at the top of the board. This part is factory tuned and must not be adjusted. It is set at an optimal operating position and if moved will drastically reduce performance.

- If mounting the board, use the mounting holes located in the center and corner of the boards. Only use plastic stand-offs and plastic screws. Using any electrically conductive materials near the board will reduce performance of the board.

- For best performance position the module boards as high off the ground as possible, in a horizontal position and keep as far away from microprocessors (or wires connected to microprocessors) as possible. High speed switching noise can interfere with reception.

- In classroom environments, many transmitters and receivers may be within communication range. When not in use, disconnect power from your project, including the module, to assure there is no chance for unintentional transmission. Transmissions from other devices can disallow other students to perform their wireless experiments.

SPECIFICATIONS

Mechanical:
Size: 1.75 Inch x 3.25 Inch x 0.4 Inch

Environmental:
Temperature: +10c to +60c

Electrical:
Power: +5 VDC +/- 0.250 VDC

Logic Levels:
Cmos: 0V = Ground = Logic 0 = Logic Low
Cmos: +5V = VDD = Logic 1 = Logic High

Current Consumption:
Transmitter: 10 Milliamp (Condition: Transmitting in switched mode)
Receiver: 15 Milliamp (Condition: Receiving in switched mode, with no output load)
Transceiver: 20 Milliamp (Condition: Transmitting in switched mode)
Note: Loading output pins will add to current consumption.

Maximum Output Loading:
Maximum load per pin is 5 milliamps.
Maximum pin loading total: 15 milliamps.

RF:
Frequency: 433.92 MHz +/- 200 KHz Typical
Transmit: Typical 1 Milliwatt
Receive: Typical -104 dbm

Can drive a single led per output pin, in switched mode with a 470 ohm resistor in series. If it is needed to drive larger loads, use NPN switching transistors.

Absolute Maximum Voltage On Any Pin: +5.25 VDC, -0.250 VDC

Web: http://www.parallaxinc.com or http://www.stampsinclass.com • Email: stamptech@parallaxinc.com
Sales / Technical Support - Tel: (916) 624-8333 Fax: (916) 624-8003
599 Menlo Drive • # 100 • Rocklin, CA 95765

PIN DESCRIPTION

PIN 1: GND: (Input) Ground pin. Connect to power supply ground.

PIN 2: +5VDC: (Input) Power Supply pin. Connect to regulated +5VDC, +/- 0.250 VDC.

PIN 3: MODE: (Input) Serial/Switch Mode Pin. The Mode Pin, determines if module will operate in switch mode or in serial mode. This pin has an internal pull down; leaving it open will be same as connecting it to ground. This pin can be left open if only switch mode is to be used. Switch mode = 0VDC, Ground, Low, Logic 0, Open. Serial mode = 5VDC, High, Logic 1. Changes on this pin can take up to 1 second to take effect. This pin can be changed on the fly, while power is applied.

PIN 4: TXD: (Input) Serial TX Data Input Pin. This pin is used to enter serial data for the transmitter to send. Serial data should be sent at 9600 Baud, 8 Data Bits, No Parity, 1 Stop Bit. Standard Serial Protocol. **CAUTION:** THIS IS NOT RS232! The module's levels are +5VDC and ground (CMOS logic levels). RS232 levels exceed +10V and -10V, levels which will damage the module. If connection to an RS232 port is desired, special circuitry is required to protect the input to the module. Continuous 9600 baud data cannot be input to this pin, at least 15 millisecond spacing must be provided between the bytes. This pin only functions in Serial Mode.

PIN 5: TXFLO: (Output) Serial Flow Control Pin. This pin is used to achieve maximum serial throughput. The module accepts data at 9600 baud but can only transmit between modules at 1200 baud. This pin is optional, but can be used to automatically pace the data for maximum efficiency. This pin should connect to the device that outputs the 9600 baud serial data to the module. When the module is ready to accept a 9600 baud byte, this pin will be low. When the module is not ready to accept a 9600 baud byte, this pin will be high. The sending device should monitor this pin to know exactly when the module is ready accept another byte for maximum data transfer efficiency. The BASIC Stamp 2 and BASIC Stamp 2sx has a built-in flow-control pin feature in the SEROUT command that works perfectly with the TXFLO pin. This pin only functions in Serial Mode.

PIN 6: RXD: (Output) Serial Receive Data Pin. This pin outputs serial data that is received by the receiver. Serial data rate is 9600 Baud, 8 Data Bits, No Parity, 1 Stop Bit. Standard Serial Protocol. **CAUTION:** THIS IS NOT RS232, its levels are +5VDC and ground, CMOS logic levels. RS232 levels exceed +10V and -10V, levels which will damage the module. If direct connection to an RS232 port is desired, special circuitry is required to protect the input to the module. This pin only functions in Serial Mode.

PIN 7: NC: (Input) NO Connection. Leave this pin open. Do not ground or terminate.

PIN 8: IN1: (Input) Switch Input 1. Internally pulled down. When left open this pin is logic low. When connected to +5VDC this pin is logic high. A logic high on this pin will send a "1" to the receive pin OUT1. This pin only functions in Switch Mode.

PIN 9: IN2: (Input) Switch Input 2. Internally pulled down. When left open this pin is logic low. When connected to +5VDC this pin is logic high. A logic high on this pin will send a "1" to the receive pin OUT2. This pin only functions in Switch Mode.

PIN 10: IN3: (Input) Switch Input 3. Internally pulled down. When left open this pin is logic low. When connected to +5VDC this pin is logic high. A logic high on this pin will send a "1" to the receive pin OUT3. This pin only functions in Switch Mode.

PIN 11: OUT1: (Output) Switch Output 1. Normally low. When transmitter IN1 is high, this pin will be high. 500 Milliseconds after IN1 on the transmitter is returned to low, this pin will return to a low state. The 4-bit address on both transmitter and receiver must match for this output pin to go high. Leaving all 4 address pins disconnected will effectively set the module to address 0. This pin only functions in Switch Mode.

PIN 12: OUT2: (Output) Switch Output 2. Normally low. When transmitter IN2 is high, this pin will be high. 500 Milliseconds after IN2 on the transmitter is returned to low, this pin will return to a low state. The 4-bit address on both transmitter and receiver must match for this output pin to go high. Leaving all 4 address pins disconnected will effectively set the module to address 0. This pin only functions in Switch Mode.

PIN 13: OUT3: (Output) Switch Output 3. Normally low. When transmitter IN3 is high, this pin will be high. 500 Milliseconds after IN3 on the transmitter is returned to low, this pin will return to a low state. The 4-bit address on both transmitter and receiver must match for this output pin to go high. Leaving all 4 address pins disconnected will effectively set the module to address 0. This pin only functions in Switch Mode.

PIN 14: ADDR1: (Input) Address Input 1. 1 of 4 address input pins. 16 different binary combinations applied to ADDR1 through ADDR4 allow unique addressing of transmitter, receiver, and transceiver pairs. This input has no effect when in serial mode (when Mode pin is connected to +5V). This pin only functions in Switch Mode.

PIN 15: ADDR2: (Input) Address Input 2. 1 of 4 address input pins. 16 different binary combinations applied to ADDR1 through ADDR4 allow unique addressing of transmitter, receiver, and transceiver pairs. This input has no effect when in serial mode (when Mode pin is connected to +5V). This pin only functions in Switch Mode.

PIN 16: ADDR3: (Input) Address Input 3. 1 of 4 address input pins. 16 different binary combinations applied to ADDR1 through ADDR4 allow unique addressing of transmitter, receiver, and transceiver pairs. This input has no effect when in serial mode (when Mode pin is connected to +5V). This pin only functions in Switch Mode.

PIN 17: ADDR4: (Input) Address Input 4. 1 of 4 address input pins. 16 different binary combinations applied to ADDR1 through ADDR4 allow unique addressing of transmitter, receiver, and transceiver pairs. This input has no effect when in serial mode (when Mode pin is connected to +5V). This pin only functions in Switch Mode.

SIMILAR COMMUNICATING MODES

Similar mode settings are when the transmitter and receiver have the same mode setting. For example the transmitter is set to serial mode and the receiver is set to serial mode. The two tables below indicate how the modules will function

Transmitter = Serial Mode	Receiver = Serial Mode
9600, 8, N, 1 serial data into the TXD pin of the transmitter and 9600, 8, N, 1 serial data out of the receiver RXD pin.	

Transmitter = Switch Mode | **Receiver = Switch Mode**

Any logic high +5VDC placed on IN1, IN2 or IN3 of the transmitter will be reflected at the receiver's OUT1, OUT2 or OUT3, respectively. Logic levels on transmitter and receiver address pins ADDR1 to ADDR4 must match.

NON-SIMILAR COMMUNICATING MODES

Non-similar mode settings are when the transmitter and receiver have opposite mode settings. For example, transmitter's Mode pin is connected to 0V (Switch Mode) and receiver's Mode pin is connected to +5V (Serial Mode)..

Transmitter = Switch Mode | **Receiver = Serial Mode**

Any logic high (+5V) applied to IN1, IN2, or IN3 of the transmitter can be read out of the RXD serial output pin of the receiver. While any of the three inputs, IN1, IN2 or IN3 are high, the transmitter will repeatedly transmit the same byte of information to the receiver. The byte contains IN1, IN2 and IN3 switch input information, in addition to the status of the 4 address pins of the transmitter. See table below.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
OUT1	OUT2	OUT3	ADDR1	ADDR2	ADDR3	ADDR4	0

IMPORTANT: OUTPUT DATA BITS AND ADDRESS BITS ARE INVERTED

Example:
The transmitter is in switch mode and +5V is applied to ADDR3 and +5V is applied to IN1. With the receiver in serial mode, the RXD pin will output the following serial byte; 01111100. The LSB, bit position "0" of the received byte is not used and is always "0". For everywhere a "1" is placed on any transmitter input pin (IN1 – IN3) the receiver will display a "0" in that bit position. For everywhere a "0" or open is applied to any transmitter input pin the receiver will display a "1" in that bit position. When a transmitter is in switch mode, it will only transmit when one of the three inputs IN1, IN2 or IN3 are brought to +5V.

Web: http://www.parallaxinc.com or http://www.stampsinclass.com • Email: stamptech@parallaxinc.com
Sales / Technical Support - Tel: (916) 624-8333 Fax: (916) 624-8003
599 Menlo Drive • # 100 • Rocklin, CA 95765

Transmitter = Serial Mode **Receiver = Switch Mode**

Serial data is input into the TXD pin of the transmitter at 9600, 8, N. Use the table below for addressing the receiver's 4 address bits and the 3 OUT bits. When a serial byte is transmitted from the transmitter to the receiver (with addresses), OUT1, OUT2 and OUT3 of the receiver will reflect the state of the bit positions in transmitted byte. The receiver reads its address pins (ADDR1 through ADDR4) to verify the received byte is intended for it. See table below.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
IN1	IN2	IN3	ADDR1	ADDR2	ADDR3	ADDR4	-

IMPORTANT: INPUT DATA BITS AND ADDRESS BITS ARE INVERTED

Example:
The transmitter is in serial mode and the following byte is sent 10110110. The receiver will compare the inverse of the 4 address bits to the settings on it's four address pins. If there is a match then it will enable it's outputs and the first three MSB bits will be inverted and sent to the OUT1, OUT2 and OUT3 (OUT2 will be high, OUT1 and OUT3 will be low). If the address does not match OUT1, OUT2 and OUT3 will not be modified. For this example, the receiver (in switch mode) will need to have ADDR1=0, ADDR2=1, ADDR3=0 and ADDR4=0. The LSB, BIT0, is a don't care bit. It can be 0 or 1 and the receiver will ignore this bit.

APPLICATIONS

O.E.M. APPLICATIONS

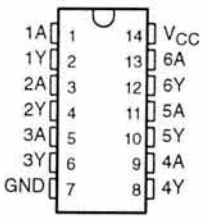
▶ For O.E.M. Design-In Guidance for the modules please contact:
RF Digital Corporation
1160 North Central Ave. Suite 201
Glendale, CA 91202
Tel: (818) 500-1082
Fax: (818) 246-9122
Web: http://www.rfdigital.com
Email: oem@rfdigital.com or info@rfdigital.com or support@rfdigital.com

- **Dependable Texas Instruments Quality and Reliability**

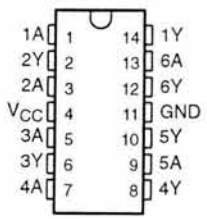
description

These devices contain six independent inverters.

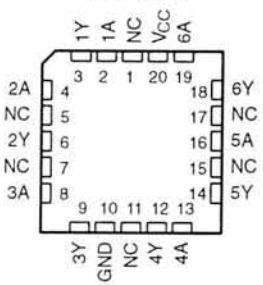
SN5404 . . . J PACKAGE
SN54LS04, SN54S04 . . . J OR W PACKAGE
SN7404 . . . D, N, OR NS PACKAGE
SN74LS04 . . . D, DB, N, OR NS PACKAGE
SN74S04 . . . D OR N PACKAGE
(TOP VIEW)



SN5404 . . . W PACKAGE
(TOP VIEW)



SN54LS04, SN54S04 . . . FK PACKAGE
(TOP VIEW)



NC – No internal connection

ORDERING INFORMATION

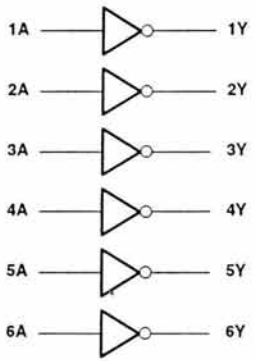
T_A	PACKAGE†		ORDERABLE PART NUMBER	TOP-SIDE MARKING
0°C to 70°C	PDIP – N	Tube	SN7404N	SN7404N
		Tube	SN74LS04N	SN74LS04N
		Tube	SN74S04N	SN74S04N
	SOIC – D	Tube	SN7404D	7404
		Tube	SN74LS04D	LS04
		Tape and reel	SN74LS04DR	
		Tube	SN74S04D	S04
		Tape and reel	SN74S04DR	
	SOP – NS	Tape and reel	SN7404NSR	SN7404
		Tape and reel	SN74LS04NSR	74LS04
	SSOP – DB	Tape and reel	SN74LS04DBR	LS04
–55°C to 125°C	CDIP – J	Tube	SN5404J	SN5404J
		Tube	SNJ5404J	SNJ5404J
		Tube	SN54LS04J	SN54LS04J
		Tube	SN54S04J	SN54S04J
		Tube	SNJ54LS04J	SNJ54LS04J
		Tube	SNJ54S04J	SNJ54S04J
	CFP – W	Tube	SNJ5404W	SNJ5404W
		Tube	SNJ54LS04W	SNJ54LS04W
		Tube	SNJ54S04W	SNJ54S04W
	LCCC – FK	Tube	SNJ54LS04FK	SNJ54LS04FK
		Tube	SNJ54S04FK	SNJ54S04FK

† Package drawings, standard packing quantities, thermal data, symbolization, and PCB design guidelines are available at www.ti.com/sc/package.

FUNCTION TABLE
(each inverter)

INPUT A	OUTPUT Y
H	L
L	H

logic diagram (positive logic)



$Y = \overline{A}$

absolute maximum ratings over operating free-air temperature range (unless otherwise noted)†

Supply voltage, V_{CC} (see Note 1) ..... 7 V
Input voltage, V_I: '04, 'S04 ..... 5.5 V
'LS04 ..... 7 V
Package thermal impedance, θ_{JA} (see Note 2): D package ..... 86°C/W
DB package ..... 96°C/W
N package ..... 80°C/W
NS package ..... 76°C/W
Storage temperature range, T_{stg} ..... −65°C to 150°C

† Stresses beyond those listed under "absolute maximum ratings" may cause permanent damage to the device. This are stress ratings only, and functional operation of the device at these or any other conditions beyond those indicated under "recommended operating conditions" is not implied. Exposure to absolute-maximum-rated conditions for extended periods may affect device reliability.

NOTES: 1. Voltage values are with respect to network ground terminal.
2. The package thermal impedance is calculated in accordance with JESD 51-7.

recommended operating conditions

		SN5404			SN7404			UNIT
		MIN	NOM	MAX	MIN	NOM	MAX	
V_{CC}	Supply voltage	4.5	5	5.5	4.75	5	5.25	V
V_{IH}	High-level input voltage	2			2			V
V_{IL}	Low-level input voltage			0.8			0.8	V
I_{OH}	High-level output current			−0.4			−0.4	mA
I_{OL}	Low-level output current			16			16	mA
T_A	Operating free-air temperature	−55		125	0		70	°C

electrical characteristics over recommended operating free-air temperature range (unless otherwise noted)

PARAMETER	TEST CONDITIONS‡			SN5404			SN7404			UNIT
				MIN	TYP§	MAX	MIN	TYP§	MAX	
V_{IK}	V_{CC} = MIN,	I_I = −12 mA				−1.5			−1.5	V
V_{OH}	V_{CC} = MIN,	V_{IL} = 0.8 V,	I_{OH} = −0.4 mA	2.4	3.4		2.4	3.4		V
V_{OL}	V_{CC} = MIN,	V_{IH} = 2 V,	I_{OL} = 16 mA		0.2	0.4		0.2	0.4	V
I_I	V_{CC} = MAX,	V_I = 5.5 V				1			1	mA
I_{IH}	V_{CC} = MAX,	V_I = 2.4 V				40			40	µA
I_{IL}	V_{CC} = MAX,	V_I = 0.4 V				−1.6			−1.6	mA
I_{OS}¶	V_{CC} = MAX			−20		−55	−18		−55	mA
I_{CCH}	V_{CC} = MAX,	V_I = 0 V			6	12		6	12	mA
I_{CCL}	V_{CC} = MAX,	V_I = 4.5 V			18	33		18	33	mA

‡ For conditions shown as MIN or MAX, use the appropriate value specified under recommended operating conditions.
§ All typical values are at V_{CC} = 5 V, T_A = 25°C.
¶ Not more than one output should be shorted at a time.

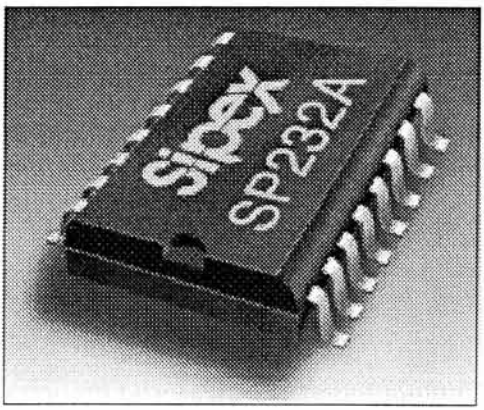
switching characteristics, V_{CC} = 5 V, T_A = 25°C (see Figure 1)

PARAMETER	FROM (INPUT)	TO (OUTPUT)	TEST CONDITIONS	SN5404 SN7404			UNIT
				MIN	TYP	MAX	
t_{PLH}	A	Y	R_L = 400 Ω, C_L = 15 pF		12	22	ns
t_{PHL}					8	15	

SP231A/232A/233A/310A/312A

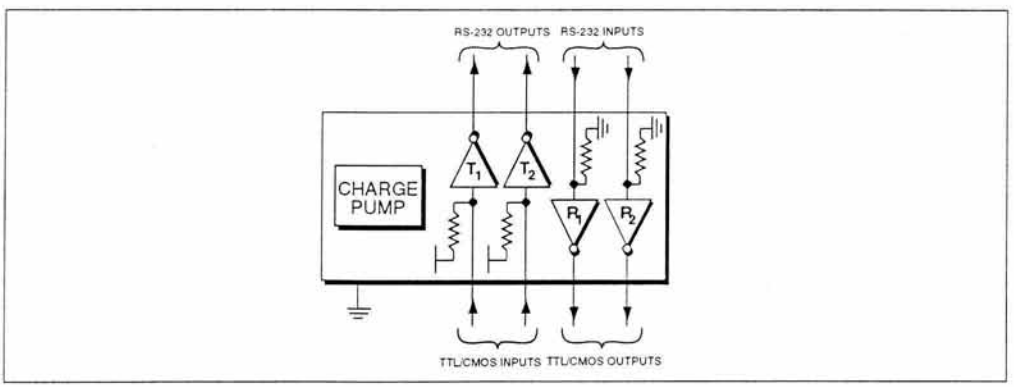
Enhanced RS-232 Line Drivers/Receivers

- Operates from Single 5V Power Supply
- Meets All RS-232D and V.28 Specifications
- Multiple Drivers and Receivers
- Small Charge Pump Capacitors – 0.1µF
- Operates with 0.1µF and 100µF Capacitors
- High Data Rate – 120kbps Under Load
- High Output Slew Rate – 10V/µs Under Load
- Low Power Shutdown ≤1µA
- 3-State TTL/CMOS Receiver Outputs
- ±30V Receiver Input Levels
- Low Power CMOS – 15mA Operation



DESCRIPTION...

The **Sipex SP231A**, **SP232A** and **SP233A** are enhanced versions of the **Sipex SP231**, **SP232** and **SP233** RS-232 line drivers/receivers. They are pin-for-pin replacements for these earlier versions and will operate in their sockets. Performance enhancements include 10V/µs slew rate, 120k bits per second guaranteed transmission rate, and increased drive current for longer and more flexible cable configurations. Ease of use enhancements include smaller, 0.1µF charge pump capacitors, enhanced ESD protection, low power dissipation and overall ruggedized construction for commercial environments. The series is available in plastic and ceramic DIP and SOIC packages operating over the commercial, industrial and military temperature ranges.



ABSOLUTE MAXIMUM RATINGS

This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operation sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods of time may affect reliability.

V_{CC} ..... +6V
V^+ ..... (Vcc-0.3V) to +13.2V
V^- 13.2V

Input Voltages
T_{IN} ..... -0.3 to (Vcc +0.3V)
R_{IN} ..... ±30V

Output Voltages
T_{OUT} ..... (V+, +0.3V) to (V-, -0.3V)
R_{OUT} ..... -0.3V to (Vcc +0.3V)

Short Circuit Duration
T_{OUT} ..... Continuous

Power Dissipation
CERDIP 675mW
(derate 9.5mW/°C above +70°C)
Plastic DIP 375mW
(derate 7mW/°C above +70°C)
Small Outline 375mW
(derate 7mW/°C above +70°C)

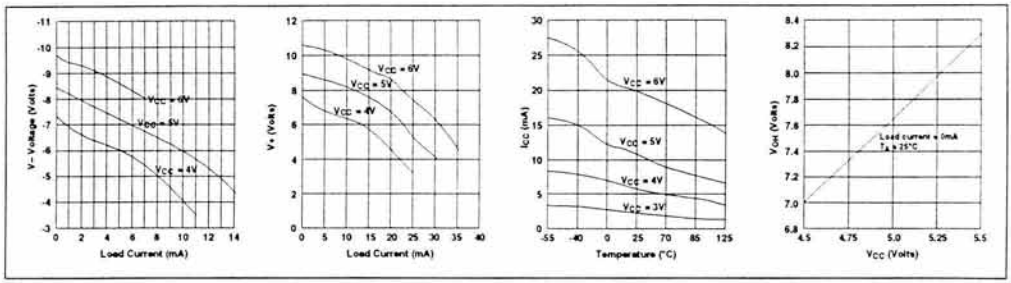
SPECIFICATIONS

V_{CC}=+5V±10%; V+=+8.5V to +13.2V (SP231A only) 0.1µF charge pump capacitors; T_{MIN} to T_{MAX} unless otherwise noted.

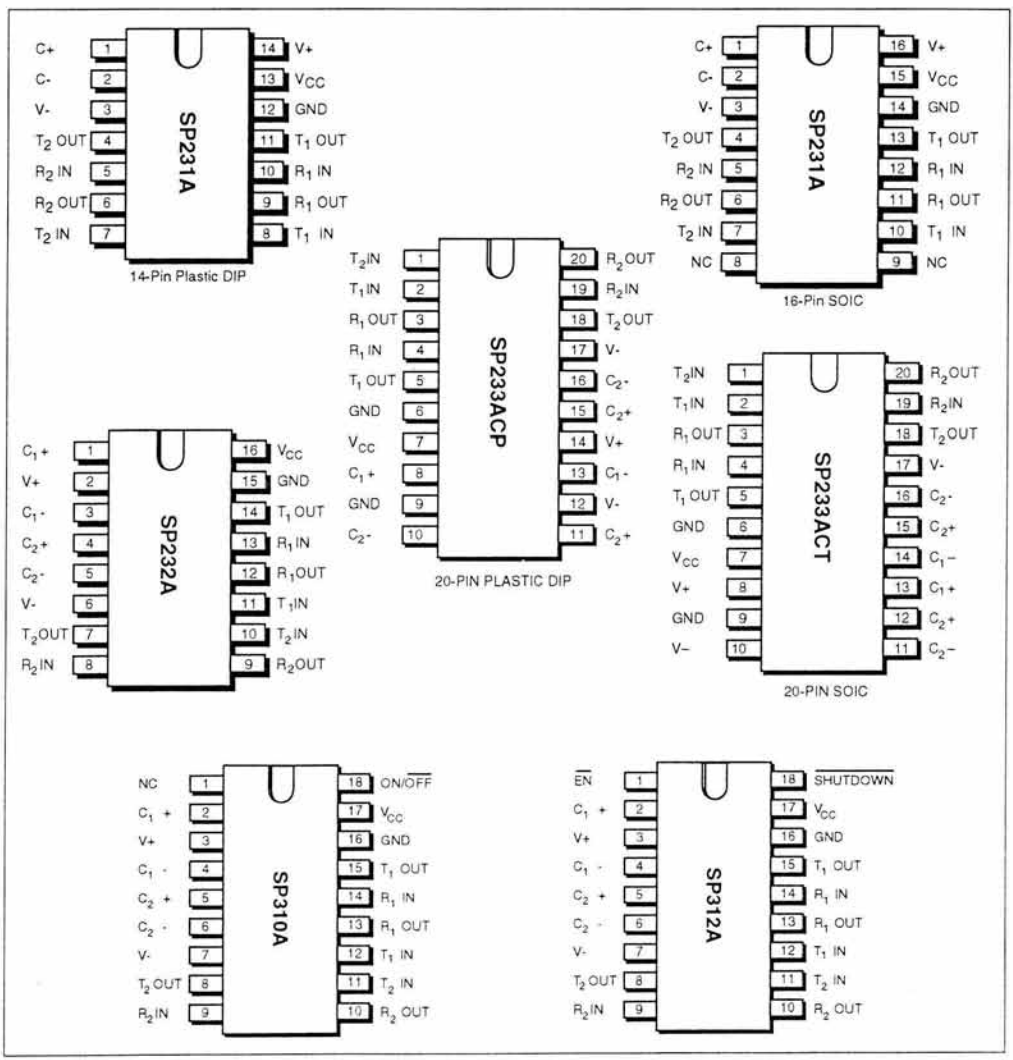
PARAMETERS	MIN.	TYP.	MAX.	UNITS	CONDITIONS
TTL INPUT					
Logic Threshold					
LOW			0.8	Volts	T_{IN}; $\overline{EN}$, $\overline{SD}$
HIGH	2.0			Volts	T_{IN}; $\overline{EN}$, $\overline{SD}$
Logic Pullup Current		15	200	µA	T_{IN} = 0V
Maximum Data Rate	120			kbps	C_L = 2500pF, R_L = 3kΩ
TTL OUTPUT					
TTL/CMOS Output					
Voltage, Low			0.4	Volts	I_{OUT} = 3.2mA; Vcc = +5V
Voltage, High	3.5			Volts	I_{OUT} = -1.0mA
Leakage Current **; T_A = +25°		0.05	±10	µA	$\overline{EN}$ = V_{CC}, 0V≤V_{OUT} ≤V_{CC}
RS-232 OUTPUT					
Output Voltage Swing	±5	±9		Volts	All transmitter outputs loaded with 3kΩ to Ground
Output Resistance	300			Ohms	V_{CC} = 0V; V_{OUT} = ±2V
Output Short Circuit Current		±18		mA	Infinite duration
RS-232 INPUT					
Voltage Range	-30		+30	Volts	
Voltage Threshold					
LOW	0.8	1.2		Volts	V_{CC} = 5V, T_A = +25°C
HIGH		1.7	2.4	Volts	V_{CC} = 5V, T_A = +25°C
Hysteresis	0.2	0.5	1.0	Volts	V_{CC} = 5V, T_A = +25°C
Resistance	3	5	7	kΩ	T_A = +25°C, -15V ≤ V_{IN} ≤ +15V
DYNAMIC CHARACTERISTICS					
Propagation Delay, RS232 to TTL		1.5		µs	
Instantaneous Slew Rate			30	V/µs	C_L = 10pF, R_L = 3-7kΩ; T_A =+25°C
Transition Region Slew Rate		10		V/µs	C_L = 2500pF, R_L = 3kΩ; measured from +3V to -3V or -3V to +3V
Output Enable Time **		400		ns	SP310A and SP312A only
Output Disable Time **		250		ns	SP310A and SP312A only
POWER REQUIREMENTS					
V_{CC} Power Supply Current		10	15	mA	No load, T_A= +25°C; V_{CC} = 5V
		25		mA	All transmitters R_L = 3kΩ; T_A = +25°C
V+ Power Supply Current ***		9	15	mA	No load, V+ = 12V, T_A=+25°C
Shutdown Supply Current **		1	10	µA	V_{CC} = 5V, T_A = +25°C

SP310A and SP312A only; * SP231A only

PERFORMANCE CURVES



PINOUT...



Receivers

The receivers convert RS-232 input signals to inverted TTL signals. Since the input is usually from a transmission line, where long cable lengths and system interference can degrade the signal, the inputs have a typical hysteresis margin of 500mV. This ensures that the receiver is virtually immune to noisy transmission lines.

The input thresholds are 0.8V minimum and 2.4V maximum, again well within the ±3V RS-232 requirements. The receiver inputs are also protected against voltages up to ±30V. Should an input be left unconnected, a 5kOhm pulldown resistor to ground will commit the output of the receiver to a high state.

In actual system applications, it is quite possible for signals to be applied to the receiver inputs before power is applied to the receiver circuitry. This occurs, for example, when a PC user attempts to print, only to realize the printer wasn't turned on. In this case an RS-232 signal from the PC will appear on the receiver input at the printer. When the printer power is turned on, the receiver will operate normally. All of these enhanced devices are fully protected.

Charge Pump

The charge pump section of the these devices allows the circuit to operate from a single +5V ±10% power supply by generating the required operating voltages internal to the devices. The charge pump consists of two sections — 1) a voltage doubler and 2) a voltage inverter.

As shown in *Figure 1*, an internal oscillator triggers the charge accumulation and voltage inversion. The voltage doubler momentarily stores a charge on capacitor C_1 equal to V_{cc}, referenced to ground. During the next transition of the oscillator this charge is boot-strapped to transfer charge to capacitor C_3. The voltage across C_3 is now from V_{cc} to V^+.

In the inverter section *(Figure 2)*, the voltage across C_3 is transferred to C_2 forcing a range of 0V to V^+ across C_2. Boot-strapping of C_2 will then transfer charge to C_4 to genrate V^-.

One of the significant enhancements over previous products of this type is that the values of the capacitors are no longer critical and have been decreased in size considerably to 0.1µF. Because the charge pump runs at a much higher frequency, the 0.1µF capacitors are sufficient to transfer and sustain charges to the two transmitters.

APPLICATION HINTS

Protection From Shorts to ±15V

The driver outputs are protected against shorts to ground, other driver outputs, and V^+ or V^-. If the possibility exists that the outputs could be inadvertently connected to voltages higher than ±15V, then it is recommended that external protection be provided. For protection against voltages exceeding ±15V, two back-to-back zener diodes connected from each output to ground will clamp the outputs to an acceptable voltage level.

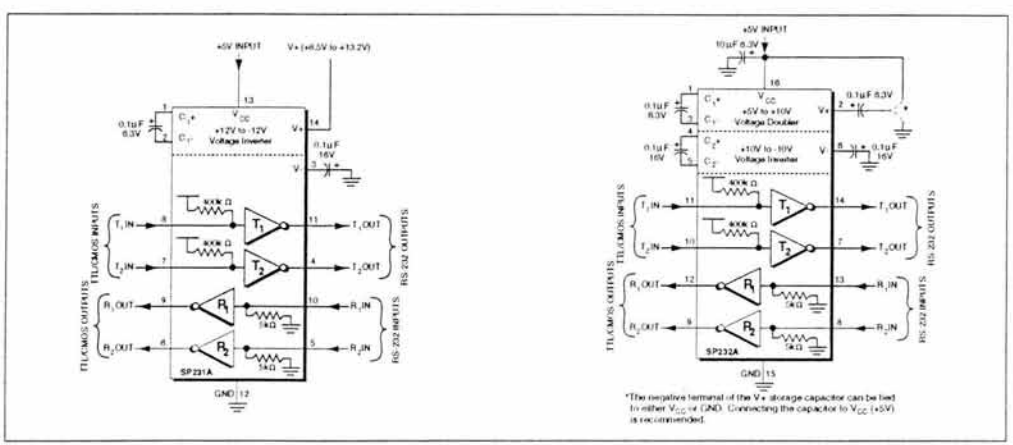


Figure 3. Typical Circuits using the SP231A and 232A.

