UNIVERSIDAD DEL VALLE DE GUATEMALA
FACULTAD DE INGENIERÍA

Implementación del bus Controller Area Network en una red de intercomunicadores con procesamiento de la señal de voz utilizando el método Delta-Adaptativa

Guatemala
2005

Implementación del bus Controller Area Network en una red de intercomunicadores con procesamiento de la señal de voz utilizando el método Delta-Adaptativa

UNIVERSIDAD DEL VALLE DE GUATEMALA
FACULTAD DE INGENIERÍA

Implementación del bus Controller Area Network en una red de intercomunicadores con procesamiento de la señal de voz utilizando el método Delta-Adaptativa

Trabajo de investigación para optar al grado académico de
Licenciado en Ingeniería Electrónica
Presentado por

José Renán Véliz Argueta

Guatemala
2005

Vo.Bo.:

(f)_____
Ing. Gonzalo Palaréa
Asesor de Tesis

Tribunal Examinador:

(f)_____
Ing. Gonzalo Palaréa

(f)_____
Dr.Ing. Manuel López

(f)_____
Ing. Julio Vásquez

Fecha de aprobación del examen de graduación: **03 de junio de 2005**

CONTENIDO

LISTA DE TABLAS

LISTA DE GRÀFICOS

LISTA DE FIGURAS

RESUMEN

En este trabajo se toma una red de intercomunicadores analógica funcional y se modifica y adapta para tener un bus de comunicaciones digital, i.e. el bus CAN. Esta transformación le confiere al sistema la característica de utilizar sólo un par entorchado de cables para enlazar hasta diez intercomunicadores por módulo central. Además, por medio de comprimir información muestreada de la señal de voz con modulación delta adaptativa se gana tiempo en el bus, el cual es vital para enviar y recibir mensajes remotos que indican solicitudes de habilitación del canal.

La construcción del sistema se basó en los buenos resultados que arrojó la previa simulación de la compresión y reconstrucción de la señal de voz con la modulación ADPCM a frecuencias desde los 16kHz hasta los 32kHz.

Se demuestra que el sistema es una solución viable gracias a la aceptable calidad de la voz reproducida en los módulos y a la capacidad del sistema de atender a breves mensajes de timbres simultáneamente así como el que la red puede abarcar distancias aceptablemente extensas.

Terminó siendo una desventaja el costo los intercomunicadores individuales, pero que se puede compensar con el ahorro en el sistema de cableado.

I. INTRODUCCIÓN

Las telecomunicaciones forman una importante área de desempeño dentro de la ingeniería electrónica. Su manifestación en sistemas de procesamiento de señales cubre aplicaciones que van desde la más simple comunicación telefónica o por intercomunicadores hasta comunicaciones globales y espaciales.

Dentro de la inmensa gama de posibles realizaciones del transporte de señales eléctricas que reproducen una voz humana o cualquier señal que corresponda a una medición sonora (con lo que el ancho de banda queda fijado), al final prevalece siempre una implementación analógica que es la encargada de actuar con el ambiente real, analógico. La realización de sistemas comerciales se basa principalmente en la conveniencia económica de implementaciones que frecuentemente yuxtaponen un sistema digital a uno analógico en estas últimas etapas de decisión.

No siempre la solución digital es la apropiada o la más económica de acuerdo a los estándares y desarrollo histórico de los sistemas técnicos con que se cuentan. Por ejemplo, las señales de cable por televisión, son aún hoy distribuidas al usuario final por medio de cables de cobre usando tecnología de segmentación frecuencial, que obviamente es un sistema de transporte de señales eminentemente analógico.

Una consideración económica para la realización técnica de la distribución de señales es la cantidad de cables por usuario necesaria a partir de una central distribuidora.

Se observa que las implementaciones de distribución requieren de recursos superfluos (en exceso o innecesarios), mientras que por el otro lado han surgido últimamente nuevas tecnologías como los protocolos I2C y CAN que teóricamente permite una reducción considerable de al menos la cantidad de cables por usuario de la distribución. Pero esto requiere un cambio en estas últimas etapas de realización de tecnología analógica a digital (no siempre ni para toda implementación es posible o deseado una reducción de cables por -usuario, por ejemplo debido a consideraciones relacionadas con la seguridad y la confiabilidad del sistema).

En este contexto se plantea la viabilidad técnica y económica de digitalización de sistemas de distribución analógica. Con el objeto de contestar a esta pregunta para sistemas comercialmente utilizados, el actual trabajo de graduación se enfoca en un sistema de intercomunicadores de un fabricante nacional. El mismo está compuesto de elementos analógicos que soportan el tráfico de señales bajo la lógica punto a punto. La etapa de interfase usuario-sistema también es analógica y no se pretende alterarla.

En el actual sistema estudiado, es físicamente necesario el enlace entre un intercomunicador y el panel central por medio de tres cables para cada puerto o usuario. El tipo de cable utilizado son los pares entorchados para telefonía que sólo se comercializan en múltiplos de dos. Esto incrementa el costo de instalación del equipo.

El objetivo general de este trabajo de graduación, consiste en demostrar que es posible integrar las nuevas tecnologías de protocolo con el fin de minimizar la cantidad de cables por usuario para ciertas aplicaciones de intercomunicación, que en la sección de métodos de trabajo será descrito. La finalidad consiste en disminuir los recursos y el costo final de implementaciones similares haciendo más eficiente el sistema. Debido a que el sistema con el que se desarrollará el prototipo es fabricado nacionalmente, se pretende llevar los resultados a niveles de exigencia industriales, proporcionando así al fabricante nacional una ventaja competitiva, que a su vez promueve el desarrollo de la industria electrónica a nivel nacional.

II. OBJETIVOS

1. Determinar la variante del método Delta Adaptativa más apropiado para la compresión de los datos de voz en el proyecto.

2. Simular confiablemente la aplicación del protocolo Delta Adaptativa sobre una señal de voz, variando sus parámetros para analizar ventajas y desventajas.

3. Construir una red CAN para un panel de control con al menos diez intercomunicadores individuales.

4. Analizar la ocurrencia de señales en un sistema de intercomunicadores para diez intercomunicadores individuales en red, contemplando señales breves y de voz.

5. Formular las exigencias de un protocolo de comunicación que permita el transporte de la máxima cantidad de datos de voz comprimida, sin que se pierdan datos de control como activación de timbres o alarmas en el sistema e implementarlo.

6. Adaptar la red digital a los intercomunicadores y al panel central a existentes introduciendo como nuevo componente el PIC18F258 en los intercomunicadores individuales y el PIC18F458 en el panel central, haciendo uso de la fuente de poder y señales de voz y direccionamiento integrados en el dispositivo analógico .

7. Digitalización de señales analógicas utilizando filtros y el módulo convertidor análogo/ digital del microcontrolador, así como conversión digital/analógica por medio de dispositivos DAC.

8. Determinar e implementar la lógica que permita multiplexar los canales de voz en el módulo central para distribuirlas según convenga al destinatario.

9. Determinar la posibilidad y frecuencia (según la velocidad y distancia) con que se multiplexa la señal de voz con señales de control.

10. Establecer prioridades para el procesamiento de cada señal según su tipo (voz, o control) y origen.

11. Establecer experimentalmente los límites superiores de distancia y velocidad del bus CAN soportados en la instalación de intercomunicadores de prueba, así como la máxima cantidad de usuarios por red.

12. Realizar un análisis de costos.

III. MÉTODOS DE TRABAJO

Se utilizará el software de desarrollo de Microchip MPLAB IDE con la aplicación MCC-18 para desarrollar el programa del PIC en lenguaje C y Assembler simultáneamente.

Se determinará por medio de simulaciones y experimentación del protocolo para compresión de voz, el método más eficaz para aplicarlo al proyecto usando como sola medida de evaluación de calidad, la inteligibilidad de la voz, variable que en última instancia depende de la pureza de la señal y la naturalidad con que sea reproducida.

Para demostración de la hipótesis se utilizará un sistema de intercomunicadores que en la actualidad, en una instalación física completa, se componen del equipo mostrado en la Figura 10.

Existen al menos dos módulos intercomunicadores individuales con capacidad de transmisión o recepción de una señal analógica de voz en forma half-duplex (e una sola dirección a la vez) y capaces de emitir señales de control. También se cuenta con al menos, un módulo o panel central que se conecta a máximo cinco módulos individuales. El panel central señaliza por medio de luces qué módulos intentan iniciar conversación con él y el usuario del mismo decide con cual establecer contacto. Desde este panel también se puede iniciar una conversación hacia cualquier intercomunicador. Es necesario indicar que la comunicación puede únicamente establecerse entre el panel central y un intercomunicador, y no entre intercomunicadores

La propuesta actual consiste en reducir la cantidad de cuatro cables por usuario, es decir en un sistema de diez usuarios, la cantidad de 4 x 10 = 40 cables a tan sólo dos para todo el sistema (topología estrella versus anillo).

Teóricamente podrían aplicarse los resultados a unos cien intercomunicadores. Por motivos de investigación práctica se fija la meta anterior a sólo diez.

Esto se pretende hacer utilizando un bus digital diferencial (no necesita tierra) muy utilizado en la industria automotriz y conocido como CAN (Controller Area Network). Este protocolo contempla la existencia de ruido en el bus y posee avanzados mecanismos de manejo de errores para contrarrestarlo. El bus está diseñado para ser manejado desde un microcontrolador, y se piensan utilizar el PIC18F258 y el PIC18F458 de la marca Microchip debido a que cumplen con la capacidad de procesamiento y módulos requeridos, así como por su relativamente reducido tamaño. Otros componentes tales como el transductor MCP2551 para enlazar el microcontrolador con el bus CAN y convertidores Digital/Análogo para la interfase con el usuario y también la modificación o inclusión de filtros se incluirán dentro del diseño de la red.

La utilización de este bus no sólo requiere de un único par de cables entorchado que enlace a todos lo intercomunicadores con la central sino que también brinda la posibilidad establecer comunicación entre intercomunicadores. El bus digital se utiliza no solo para enviar o recibir una señal de voz digitalizada sino que simultáneamente pueden recibirse o enviarse otro tipo de datos, entre ellos el estado de interruptores con algún propósito definido (botón de pánico, alarmas, etc.).

IV. MODULACIÓN PCM

El método más conocido y comúnmente utilizado para muestrear una señal analógica es el de PCM (Pulse Code Modulation). La salida de un codificador como éste es un tren de pulsos seriales equitativamente dimensionados que representa a la señal.

En su modelo más básico esta modulación se ocupa de muestrear la voz a la mínima frecuencia posible, i.e. 8 kHz u 8 kbytes/seg. ya que la voz humana tiene un ancho de banda de 4kHz (sólo frecuencias significativas ver Figura 1)

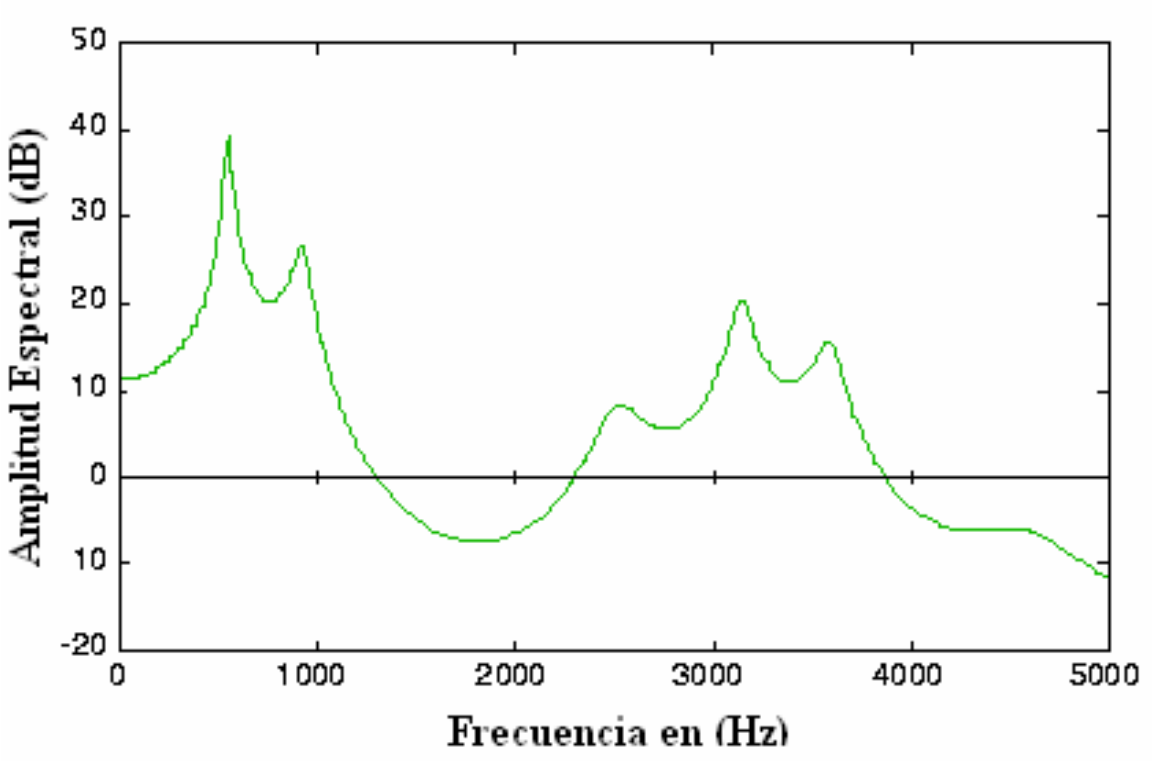


Figura 1: Función de transferencia del tracto vocal

De cada muestra se obtiene un byte o palabra de 8 bits, lo que se traduce en una frecuencia de muestreo t de 64k bits/seg. Se requeriría de un canal con un ancho de banda de igual a dicha velocidad de muestreo como mínimo para trasmitir la señal en tiempo real aproximadamente. Si se transmitiesen los bits en forma de unos y ceros lógicos se tendría en el canal una señal portadora con componentes DC altos y de frecuencias bajas como para que recorriera inalteradamente a largas distancias.

Entonces la señal PCM es modulada sobre una portadora de alta frecuencia utilizando una forma de onda digital continua como por ejemplo ASK(Amplitude Shift Keying), FSK (Frequency Shift Keying).

Estas modulaciones requieren componentes electrónicos de alta frecuencia, que son costosos y no se justifican para la presente aplicación. Además la aplicación requiere de un bus donde se permita el arbitraje y no donde el canal sea utilizado únicamente por un usuario a la vez.

En cualquier modelo de modulación lo que se busca es la manera tener un canal transmisor lo mas eficiente posible y conferir a la señal transmitida la propiedad de ser robusta a los errores.

V. EL BUS CAN

La idea de implementar el bus CAN (Controller Area Network) proviene de la necesidad de utilizar una modulación que alcance largas distancias y sea robusta a los errores. El bus CAN utiliza una modulación que envía datos binarios en forma de niveles lógicos no absolutos como 0 y 5 Voltios, los cuales requieren de una tierra como referencia sino que diferenciales, es decir donde se envían dos señales idénticas pero invertidas y centradas con respecto a un voltaje común que de variar altera la referencia de ambas. Como los alambres sobre los que viajan las señales son un par entorchado el ruido afecta a ambas en la misma magnitud y aunque la referencia varíe no lo hace la diferencia entre ambas, que es lo que se mide en los extremos. Además el sistema de transmisión no requiere de un reloj que proporcione sincronía ya que los flancos señal de datos contienen de forma codificada la señal de reloj. Específicamente el tipo de reloj codificado utilizado en CAN es el conocido como NRZ (Non-Return-To-Zero). Además el módulo por sí sólo implementa el concepto de PLL (Phase Locked Loop) para sincronizar el reloj local con el remoto.

Los estados diferenciales del bus que corresponderían a estados lógicos digitales se conocen como estado recesivo (hay diferencia de mínimo 1Voltio) y dominante (no hay diferencia).

En CAN la información acerca del nodo con el que se desea establecer comunicación está contenida en cada mensaje y se le denomina identificador, el cual a su vez indica la prioridad del mensaje (a menor magnitud , mayor prioridad). Cada nodo posee filtros que le permiten obviar mensajes o bien escucharlos, es decir que es selectivo.

El bus tiene capacidad de Arbitraje conocida como CSMA/CD (Carrier Sense Multiple Access/Collision Detection with non destructive Arbitration). Cada nodo que desea transmitir primero revisa que el bus esté libre y al transmitir se convierte en maestro del bus. Si un segundo nodo desea transmitir al mismo tiempo se evita una colisión por medio de arbitrar el bus a manera de bits, los bits dominantes se sobreponen a los recesivos

Cada nodo inspecciona que en el bus exista la señal que él mismo está colocando, de no ser así el nodo sabe que ha perdido el arbitraje e intentará tomar el bus de nuevo más tarde, pero la información puesta en el bus por el nodo que ganó el arbitraje no se pierde, de allí que el arbitraje se no destructivo.

CAN provee la posibilidad de enviar tramas de datos o bien solicitarlas. Una trama de datos pude tener longitud variable de 0 a 8 bytes de datos. Una Requsición Remota de Datos (RTR = Remote Transmisión Request) es un tipo de mensaje que no contiene datos por defecto y por tanto es mucho más corto.La trama de un mensaje CAN con datos contiene como máximo quince bytes o 120 bits, como lo demuestra la siguiente tabla:

Tabla 1: Contenido de una trama CAN

# bits	**Dato**
1	Stara
11	Identificador
1	RTR
3	Reservados
4	Tamaño Datos
64	Datos
15	CRC (Cyclic Red. Check)
1	CRC Del
2	Acknowledge
7	Fin de Trama
8	Transmisión
8	Espacio entre Frames
120	**Σ Trama CAN**

Existen cinco Tramas:

1) Standard Data Frame (15 bytes)

2) Extended Data Frame

3) Error Frame

4)RTR Frame (Remote Trans. Request)

5) Overload Frame

Debe notarse que sólo el (l64/120)*100 = 53%

de la trama son datos, el resto es encabezado.

El total de 120 bits equivalen a 15 bytes para una trama CAN completa. Como un máximo de 8 bytes de datos es transmitido por trama, se tiene que la eficiencia del bus medida en términos de bytes de datos con respecto a bytes transmitidos es del 53% .A una velocidad de bus de 125kbits/seg., la eficiencia se puede interpretar como que si se estuviesen transmitiendo solamente datos a una velocidad de 66kbits/seg. Así se podrían enviar tramas de muestras codificadas a 8 bits sin comprimir.

Es decir que tan sólo tomando muestras analógicas, cuantificándolas, digitalizándolas y enviándolas continuamente y usando el bus CAN a 125kb/s es como tener una velocidad de muestreo de 53%*125 = 66kb/s. Como el estándar de muestreo para PCM es de 64kbits/s esto equivaldría a tener un bus CAN con velocidad de 64k / 53% = 121 kb/s, lo que significa que una trama ocupa el bus por 0.991ms. Entre trama y trama sobrarían 1/121k-1/125k = 26.4us. Este tiempo no alcanza para introducir ningún tipo de trama en el bus, ni siquiera de mensajes remotos. El cálculo indica que el ancho de banda libre en el bus sería de un 0.0264/(0.991+0.0264)*100 = 2.6%.

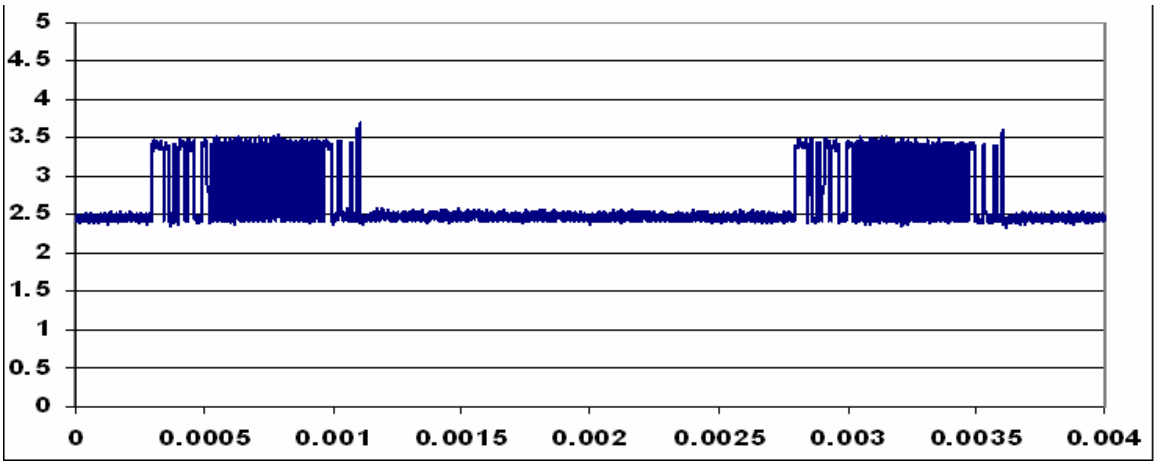


Figura 2: Dos tramas del Bus CAN a 125kb/s. Período tramas = 2.5ms, trama = 0.9ms, Entre Tramas = 1.6ms,

Del pensamiento anterior surge la necesidad de comprimir los datos acerca de las muestras de una señal analógica digitalizada. Así se envían paquetes de datos en forma espaciada, liberando el bus por un tiempo sustancial entre tramas y permitiendo una aceptable reconstrucción de la señal.

De acuerdo a la literatura el bus es funcional desde distancias de 40m a 1Mbits/seg. hasta 10km a 5kbits/seg. El factor principal limitante de las longitudes alcanzadas son los retrasos en la propagación de las señales transmitidas. Para la aplicación actual se escogió una velocidad de bus de 125kbits/seg., porque con ella teóricamente se pueden cubrir distancias de hasta 500m.

En cuanto a la cantidad de nodos que se pueden sumar al bus, utilizando los transductores apropiados, esta es teóricamente de alrededor de 115 nodos. El factor limitante con respecto a esta cantidad es la resistencia de carga que estos presentan y la corriente que son capaces de entregar los “transceivers”.

A continuación se explica cómo el método Delta-Adaptativa comprime la señal de datos para ocupar aproximadamente 900useg. del bus cada 2.50 ms (muestreo a 23kHz) , lo que provee una ventana de por lo menos 1.6ms entre trama y trama de datos, durante la cual módulos esclavos pueden enviar y el maestro recibir mensajes remotos (RTRs). Así el bus se encuentra libre un 1.6ms/2.5ms = 64% del tiempo en que se usa para enviar paquetes de voz.

VI. LOS MÉTODOS DE COMPRESIÓN DELTA

A. Modulación Delta

Este método de compresión de datos digitales entra dentro de la categoría de las Modulaciones Delta. Con éstas se pretende eliminar la redundancia de bits en un flujo de datos codificados con PCM. Por naturaleza son menos sensitivas a errores en el canal. La modulación delta utiliza tan sólo un bit para cuantificar la señal analógica. La salida de la codificación delta es siempre un uno o un cero, dependiendo de si la muestra presente es mayor o menor con respecto al valor mostrado en un período anterior. El cero se interpreta como una disminución en una magnitud fija de la nueva señal con respecto a la anterior y un uno es visto como un aumento en la misma magnitud. Es decir que el diferencial delta, conocido como paso delta es constante (ver Figura 3).

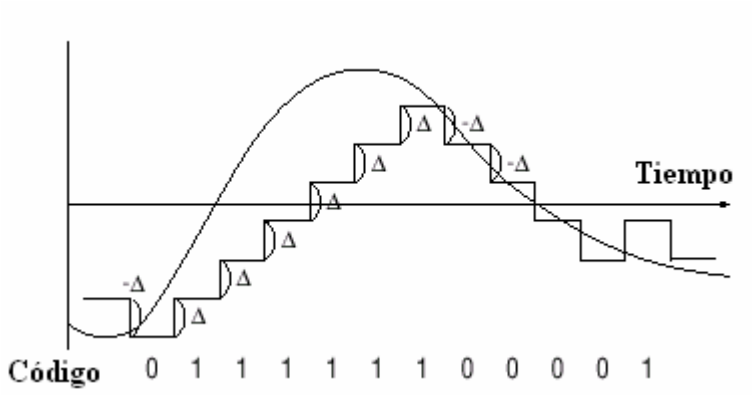
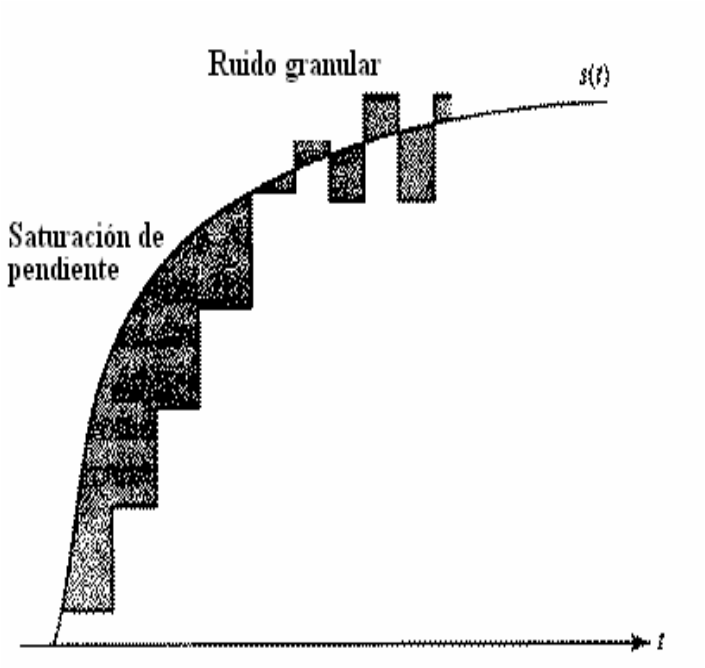


Figura 3: Modulación Delta

La compresión de los datos está proveída por el siguiente hecho; si una muestra de la señal codificada con PCM toma k unidades de tiempo en transmitirse, la misma muestra codificada con el algoritmo delta toma 1/k unidades de tiempo en enviarse, donde k es el número de bits utilizados para cuantificar la muestra.

Una de las ventajas de la modulación delta es que es menos propensa a las transiciones entre unos y ceros en su salida codificada con respecto a la de PCM. Sin embargo este algoritmo posee dos grande desventajas. Debido a que el paso delta ya sea en aumento o disminución es constante, si éste es muy pequeño se obtiene una saturación de la pendiente que describe la tendencia de la señal reconstruida.

Es decir que la señal real aumenta o disminuye con una pendiente mayor a la que el paso delta es capaz de proveer aún si sólo cambia en la misma dirección (sólo crece o sólo disminuye). Por otro lado, si el paso delta es muy grande, los cambios pequeños en la señal original son interpretados, por decirlo así, con pasos desmedidos en la señal reconstruida. Este fenómeno se conoce como granularidad en la señal. Es importante entonces escoger acertadamente el valor delta a utilizar (ver Figura 4).

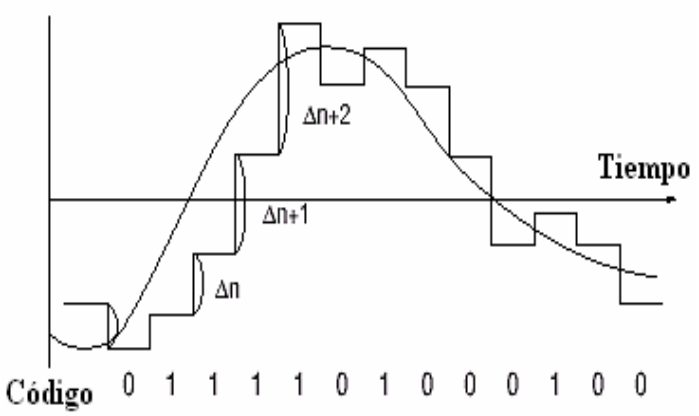


Para evitar la aparición de las dos ocurrencias desventajosas ya mencionadas es normalmente necesario muestrear la señal original, según la literatura a una frecuencia entre dos o cuatro veces mayor a la de PCM, según la aplicación. Para la voz esto equivaldría a muestrear entre 16 y 32 kHz.

Figura 4 :Saturación de pendiente y granularidad por Modulación Delta

B. Modulación Delta Adaptativa

En esta forma de modulación se aplica el mismo concepto que en la modulación delta con respecto a cuantificar la diferencia entre muestras utilizando un solo bit. La diferencia radica en que este método posee un paso delta variable. El algoritmo evalúa la tendencia de la señal, es decir, si esta procura crecer o disminuir antes de decidir la magnitud del paso delta a utilizar.



La Modulación Delta Adaptativa se divide en dos metodologías básicas. La primera es la cuantización adaptativa prealimentada.

Figura 5: Modulación Delta Adaptativa

(AQF = Adaptive Quantization Forward). Esta forma requiere de almacenar un grupo de muestras, codificarlas y de acuerdo a sus diferencias definir el paso a utilizar. Es deficiente en el sentido de que no ahorra ancho de banda al tener que enviar el grupo de datos codificado y además la dimensión del paso delta. Por otro lado confiere un retraso considerable a la señal reconstruida debido a la necesidad de acumular muestras para evaluarlas.

Existe también la Cuantificación Delta Retroalimentada (AQB =Adaptive Quantization Backwards) que es más común debido a las deficiencias de la anterior. Aquí no se acumulan muestra sino que se codifican las diferencias igual que en la modulación delta, a un bit. Sin embargo sólo se transfieren los códigos de diferencias entre muestra original y reconstrucción anterior.

El paso delta se determina en la reconstrucción evaluando la tendencia de las diferencias. El número de diferencias anteriores que se evalúen determina la velocidad con que el paso delta crece.

Puede, por ejemplo, definirse que si el código anterior o k códigos anteriores es/son iguales al presente el paso sea aumentado en una constante, y si no lo es/son, el paso, disminuye en esa misma cantidad, siempre cuidando no sobrepasar los límites de un paso máximo o mínimo previamente definidos.

Debe notarse que el paso puede o no cambiar con la reconstrucción de cada muestra pero siempre se le suma o resta a la muestra anterior. Esto implica que para una señal original constante se obtiene una reconstruida con una oscilación alrededor de la constante en magnitud del paso mínimo definido.

La codificación Delta Adaptativa es muy útil para codificar voz a tasas de transmisión medios. La ITU (Internacional Telecommunications Union) propone el estándar de codificación de voz para telefonía ADPCM G.726 a una velocidad de 32 kbits/seg. Llama la atención que la calidad de la voz reproducida por este algoritmo es evaluada con un 4.1 en una escala MOS (Mean Opinion Scores) y una complejidad medida en términos de la capacidad de un procesador, i.e. 2 MIPS (Mega Instrucciones por segundo).

No existe una medida completamente objetiva que evalúe la calidad de la reproducción de voz más que la MOS. En comunicaciones de voz, especialmente en telefonía por Internet el MOS provee una medida numérica de la calidad e inteligibilidad de la voz humana. El sistema usa tests subjetivos (opiniones en forma de puntajes de desde 0 = malo hasta 5 = excelente) que se promedian matemáticamente para obtener un indicador cuantitativo del desempeño del sistema.

VII. DETERMINACIÓN DE LA FRECUENCIA DE MUESTREO

La frecuencia de muestreo propuesta por la literatura y algunos estándares de telefonía para el algoritmo ADPCM es de 32kbits/seg. Sin embargo estudios de trabajos de investigación anteriores afirman la claridad de la voz muestreada a 16 kbits/seg. con el mismo método es buena y que éste es el sistema más recomendable de para enviar señales de voz sobre un canal con ancho de banda limitado. El microcontrolador a utilizar posee una capacidad de 10 MIPS. Con esto y debido a las limitantes de su módulo convertidor análogo digital la velocidad de muestreo es teóricamente de 32kMuestras/seg., si esa fuese su única tarea.

Sin embargo, este integrado debe realizar también las funciones de comprimir los datos, enviarlos por el bus CAN, recibir Mensajes Remotos del bus CAN, así como generar y atender señales de control locales. De allí que con todo el código implementado su capacidad de muestreo quede reducida a 23 kilo-muestras/s en el caso del microcontrolador maestro. El microcontrolador esclavo podría tomar muestras ligeramente más rápido, pero se lo implementó a la misma velocidad de muestreo que el módulo maestro 23kHz. El decremento de la frecuencia va de la mano con la calidad de la voz reproducible. Como este factor se evalúa subjetivamente, el juicio definitivo acerca de si el algoritmo y su velocidad de muestreo son los apropiados lo da el usuario que con su opinión define se la señal recibida es clara e inteligible.

Para predecir la eficiencia del algoritmo ADPCM a 25kHz se elaboró una simulación con Excel y Visual Basic que arrojan una fiel reproducción de los resultados que se obtendrían en la realidad. Se simula la codificación de una señal de entrada senoidal a 500Hz con el algoritmo propuesto para CAN, en el que sólo la primera muestra es de 8 bits y las siguientes codificadas con ADPCM de un bit. La señal de salida se hace pasar por un filtro pasa-bajas de 4 polos. Una función senoidal de 4 Vp-p se muestrea a 32kHz, luego a 25kHz después a 20kHz y finalmente a 16kHz. Las gráficas resultantes se encuentran en el apéndice numeradas de la 1 a la 4. La Gráfica 5 representa la función de transferencia del filtro pasa-bajas utilizado utilizando convolución. La señal de entrada es la superior, la modulada es la central y la inferior es la filtrada con el filtro simulado.

VIII. LA RED DE INTERCOMUNICADORES

La red de intercomunicadores consiste en un módulo maestro capaz de manejar hasta 15 módulos esclavos. El módulo maestro se compone de dos partes. La primera es la circuitería analógica de “switcheo” para encender y apagar luces que indican el estado de cada esclavo, es decir si tiene el canal o bus CAN habilitado, deshabilitado o si solicitó habilitación. Otra parte del circuito analógico provee la potencia, es decir voltaje de 5V y contiene el micrófono que recibe la voz y la convierte en un voltaje analógico y amplifica así como la bocina que recibe una señal analógica reconstruida, ya sea de voz o del timbre. El timbre también se genera analógicamente con un integrado LM555.

La otra parte del maestro es un circuito digital cuyo eje es el microcontrolador PIC18F458. Este corre a 10MIPS y tiene a su cargo la conversión análogo digital de la señal de voz proveniente de un filtro pasa-bajas a 5kHz y a su vez del micrófono. También se encarga del envío en forma serial hacia un convertidor digital análogo (MCP4921) del la señal de voz digital proveniente de algún módulo esclavo a través del bus CAN. Maneja el bus CAN definiendo quien puede enviar paquetes de voz. generación del BCD (Binary Coded Decimal) genera el timbre cuando hay una solicitud del canal y la recepción del BCD que indica la dirección del canal que debe habilitarse.

El módulo esclavo es principalmente un microcontrolador, el PIC18F258 que corre a 10MIPS tiene la misma memoria de programa, RAM y ROM que el 18F458 pero simplemente menos puertos de entrada y salida. En este módulo se obtiene la señal de voz de un micrófono “Electret” de capacitor, se amplifica y pasa por un pasa-bajos para convertirse de análogo a digital en el microcontrolador. Luego comprime con el algoritmo ADPCM y se envía por CAN.

Un mensaje recibido en CAN es escuchado y evaluado en cuanto a identificador para determinar si el mensaje le corresponde y ponerlo en el convertidor digital análogo para reconstruir la señal, filtrarla y llevarla a un amplificador de audio con el que reacciona una bocina de 8 Ohmios o bien de 16 Ohmios y 0.5Watts.

IX. ARBITRAJE DEL BUS EN SOFTWARE

El bus CAN provee un tipo de arbitraje robusto en el caso en que el uso del bus no sea casi permanente. Si un solo nodo esta constantemente enviando paquetes de voz por ejemplo, difícilmente otro nodo vaya a poder enviar sus propios paquetes de voz aún si existe arbitraje porque van a haber colisiones constantemente. Esto se debe a que el tamaño de los paquetes de voz en el bus (900us) es más de la mitad del tiempo que dura libre el bus (1.6ms) y la probabilidad de una colisión es mayor. Por otro lado mientras el bus está libre el nodo que envía paquetes se encuentra creando el siguiente paquete y no puede ni debe reconstruir paquetes de voz.

De allí surge la necesidad de impedir el envío de paquetes de voz simultáneamente entre cualquier nodo (Maestro o Esclavo). Mientras el bus está ocupado por un esclavo o maestro solamente cabe la posibilidad de que los esclavos envíen mensajes remotos (excepto el que ocupa el bus enviando paquetes de voz), y que el maestro los escuche o bien escuche paquetes de voz simultáneamente. Si el maestro ocupa el bus sólo puede escuchar mensajes remotos, porque los esclavos no pueden enviar paquetes de voz. Esto se logra impidiendo la generación de paquetes de voz y su acceso al bus a menos que el canal este habilitado para ese nodo esclavo y hayan pasado 0.4ms desde la última vez que el bus estuvo ocupado, lo cual garantiza que en el momento presente el bus sí está libre.

Se define que todas las tramas con paquetes de datos contienen 8 bytes de datos. Sólo los mensajes remotos considerados como señal de timbre carecen de datos. De esta forma una trama de voz contiene 15 bytes mientras que una de requisición sólo 7. Esto permite que las requisiciones puedan intercalarse en los tiempos libres del bus además de que tengan la prioridad de adquirir el bus por el uso de los identificadores de menor magnitud. Los mensajes remotos tienen asignadas los primeros quince identificadores. Los paquetes de voz se etiquetan con el identificador del microcontrolador local más veinte unidades. Así un mensaje remoto supera en prioridad a uno de voz y el módulo maestro escuchará, si éste no intenta sobreponerse a otro menaje remoto con identificador de inferior magnitud.

X. PRE-FILTRADO Y POST-FILTRADO DE LA SEÑAL DE VOZ

La señal proveniente de un micrófono debe de ser filtrada a 4kHz para garantizar que no hay componentes frecuenciales más haya de ese valor, los cuales el algoritmo de compresión no contempla. De por sí el algoritmo es deficiente para perseguir señales más allá de 1 kHz pero por otro lado si no se eliminan frecuencias superiores en el dominio de la frecuencia ocurre un fenómeno conocido como “aliasing” y es que como los espectros se repiten cada dos veces la frecuencia de muestreo, si se muestrean señales con frecuencias de superiores a la mitad de la de muestreo, dichos espectros se traslapan y la señal reconstruida queda distorsionada irreversiblemente. Otra razón para filtrar la señal antes de cuantificarla digitalmente es que el microcontrolador genera ruido en la fuente que frecuentemente en cualquier etapa de amplificación se introduce en la señal. Este ruido queda eliminado con el pre-filtrado.

Lo anterior justifica filtrar la señal antes de muestrearla. La razón de filtrar la señal reconstruida después del convertidor digital/análogo es más obvia. Como las muestras reconstruidas son proveídas a dicho convertidor con una frecuencia de 23kHz, los niveles analógicos alcanzados difieren entre sí por al menos la mínima unidad o error de conversión (1bit = 20mVolts). Estas diferencias se dan abruptamente en forma de escalones, lo que introduce en la señal componentes frecuenciales altas que en realidad no se encuentran en la señal original y son audibles pero deben ser eliminadas con un filtro pasa-bajas también a 4kHz.

El filtrado se realiza por medio de los circuitos integrados MAX7426 que son filtros pasa-bajas de forma elíptica y con cinco polos, lo cual significa que para componentes frecuenciales de al menos una década superiores a la frecuencia de corte se tiene que su pérdida es de al menos -100dB. Para esta aplicación, los filtros se configuran colocando un capacitor de 33pf en su entrada de reloj, para que el reloj interno genere oscilaciones a 500kHz que se traducen en una frecuencia de corte de 5kHz. La fórmula utilizada es:

$$fc\ (kHz) = 17.5x10^{-3}/Cosc(pF)$$

La función de transferencia del filtro está descrita por la Gráfica 6**.**

XI. EL MÓDULO MAESTRO

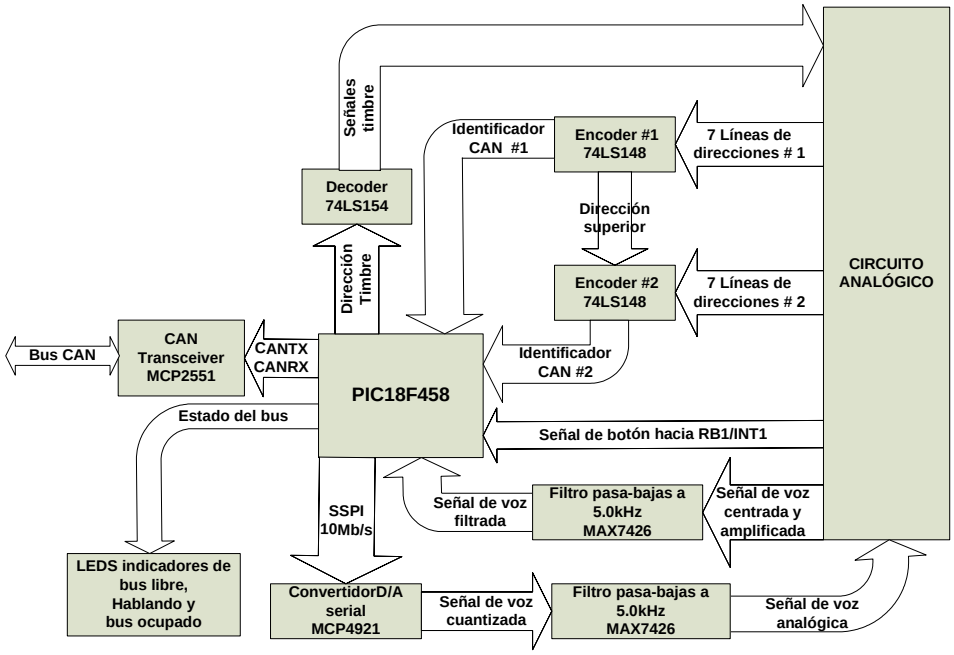


Figura 6: Diagrama de las interconexiones entre el PIC18F458, sus periféricos y el circuito analógico y el bus CAN

La forma en que se dispuso en software de los pines del microcontrolador PIC18458 se encuentra descrita en la Tabla 1. Los diagramas de la Figura 11 y 12 describen cómo cada pata del microcontrolador se conecta a los demás integrados del circuito al igual que con el circuito analógico.

El microcontrolador escogido para el módulo maestro tiene varias funciones a su cargo:

1. Leer, del de un puerto la dirección proveída por dos decodificadores de prioridad y que se traduce en el identificador del módulo al que se le enviarán mensajes
2. Determinar por medio de flancos el estado de la señal de botón y así decidir si en estado de envío de paquetes de voz o recepción de mensajes de voz o remotos.
3. Utilizar su convertidor A/D para digitalizar la señal de voz, centrada en 2.5 V, amplificada entre 0 y 5 V y prefiltrada.

4. Recibir mensajes remotos del módulo CAN y poner su identificador en un puerto para que sea utilizado por el decodificador de señales de timbre.
5. Recibir mensajes empaquetados de voz y descomprimirlos para luego enviarlos a un convertidor D/A serialmente, para luego ser post-filtrados y enviados en forma de señal de voz reconstruida al circuito analógico.
6. Mostrar con tres LEDS el estado del bus (libre u ocupado) y del botón de hablar.

El diagrama de flujo del programa principal para el microcontrolador PIC18F458 está en la Figura 14. El orden de atención a las interrupciones para el mismo está en la Figura 15 y 16.

Se asume que el circuito analógico recibe e interpreta correctamente las señales que el circuito digital le provee así como que envía adecuadamente las señales que el digital necesita. Las funciones del circuito analógico son:

1. Proveer una señal de voz libre de ruido centrada en 2.5V y amplificada entre 0 y 5V.

2. Entregar diez líneas de direcciones, donde el estado habilitado se indica con un cero lógico.

3. Entregar la señal del botón con cuyo flanco se activan la transmisión/recepción y datos.

4. Recibir, amplificar y entregar potencia a la señal de voz post-filtrada que se pondrá en la bocina.

5. Recibir las señales de timbre del decodificador y generar la señal de timbre.

A. Algoritmo de compresión

El Algoritmo de Compresión es el ADPCM a 23kHz . Como variante del algoritmo y se introdujo el hecho de que en el primer byte de un paquete de datos se encuentra una muestra digital a 8 bits de la señal analógica muestread. Esta muestra sirve de referencia para la suma del paso delta a aplicar sobre los siguientes 56 bits de datos diferenciales. De esta forma cada paquete de voz contiene 57 muestras, una de a 8 bits y el resto de a bit.

El algoritmo comienza tomando la primera muestra y colocándola en la primera posición del vector de datos CAN y en un registro de referencia. La siguiente muestra se compara con el registro de referencia y si es mayor que éste se coloca un uno en el registro de muestras diferenciales, de lo contrario se coloca un cero.

Luego se consulta el estado de los cuatro bits anteriores para determinar el paso delta que empieza con un valor de 1 bit (20mV). Si los cuatro son idénticos al presente esto se entiende como una tendencia al incremento por lo que el paso se duplica, éste vale ahora 2 bits. Si en la secuencia de bits anteriores existe tan sólo uno que es diferente al presente el paso se reduce directamente a 1 bit.

Si después de varias muestras y comparaciones el paso delta alcanza el valor de 32 bits (0.64Volts) y al siguiente paso le corresponde aumentar porque continua habiendo una coincidencia entre los cuatro últimos bits codificados, el paso delta permanecerá igual. En el caso de una discrepancia entre bits (la señal ya no es monótonamente creciente o decreciente) el paso regresa a ser de magnitud unitaria. El dimensionamiento del paso delta en el módulo que comprime los datos de la señal es necesario para llevar control de cómo el módulo receptor va a reconstruir los datos de voz con respecto a la señal fuente. Es decir que los eventos a partir de que se compara el resultado del bit actual con respecto a cuatro anteriores constituyen el algoritmo de descompresión. Sucede entonces que el módulo del que proviene la señal, también la reconstruye pero no coloca la en un convertidor D/A, en cambio el módulo receptor sí.

Al final de cada comparación entre bits, el paso se suma o resta al registro de referencia para actualizarlo. Al igual que el resultado del convertidor A/D, la magnitud del registro de referencia no puede sobrepasar el valor de 255 (5 Volts) ni decrementar más allá de cero. Cada 8 bits codificados se colocan en un byte del vector de datos CAN. Cuando se tienen 8 bytes de muestras, se un paquete de voz completo en el bus CAN. La conversión análogo/digital el primer dato que encabeza la siguiente trama codificada no se inicia sino hasta que el paquete anterior ha sido puesto en el módulo CAN de transmisión y no necesariamente transmitido.

Una descripción más detallada del algoritmo la provee la Figura 18.

B. Lectura de dirección

El identificador de cada mensaje CAN a enviar está dado por el registro en que se guarda la dirección leída directamente de uno de los puertos del microcontrolador (PuertoD). Hardware externo, i.e. dos codificadores de 8 a 3 líneas (circuito integrado 74LS148) colocados en cascada. Estos se encargan de determinar cuál de los 10 botones de direccionamiento en el panel de control ubicado en el circuito analógico se presionó de último y con esto generar el número BCD que arroja directamente la dirección sobre 6 líneas del Puerto D. Como los codificadores son de prioridad, aún si se presiona más de un botón a la vez los codificadores arrojan la dirección de aquel cuya dirección es menor. Que estén colocados en cascada tiene el efecto de no permitir que las tres superiores líneas de dirección se activen, es decir que no reconoce las ocho últimas direcciones a menos que ninguna de las primeras ocho se encuentre seleccionada. (Ver Circuito maestro parte 2).

C. Recepción de mensajes remotos o requisiciones remotas

El módulo maestro es capaz de recibir mensajes remotos en cualquier momento, ya sea si esta escuchando y reconstruyendo paquetes de voz de cualquier otro módulo o bien transmitiendo él mismo paquetes de voz. Esto es posible gracias a que el bus está libre 64% del tiempo en que se usa para enviar paquetes de voz y también a que la trama de las requisiciones remotas es más corta que la de un paquete de voz.

Además las requisiciones remotas tienen asignado un identificador de mayor prioridad al de los paquetes de voz. Esto garantiza que el maestro siempre los escuche y que no pierdan el arbitraje ante un paquete de voz.

Una vez el maestro recibe una requisición remota, éste toma su identificador y lo interpreta como una dirección BCD (Binary Coded Decimal) de cuatro bits y luego la coloca en los cuatro bits menos significativos del Puerto B del microcontrolador. A partir de allí son leídos por las líneas del circuito integrado 74LS154. Este es un decodificador de 4 a 16 líneas. La dirección puesta en su entrada se traduce a poner en estado de uno lógico precisamente una de las 16 salidas a las que corresponde la dirección.

Estas salidas se conectan al circuito analógico del maestro y en aquellos puntos en que el estado lógico uno ocasiona que se encienda un LED rojo justo al lado del botón que habilita el bus para un esclavo específico. A la vez que el LED se ilumina, el mismo circuito analógico activa el integrado LM555 para que genera una onda que suena como un timbre en la bocina del modulo maestro. Este timbre se produce durante 50ms, suficiente para ser escuchado. Durante el tiempo en que se produce un sonido de timbre no se puede generar uno más ocasionado por otra requisición remota, pero esto no representa un problema porque no es probable que las requisiciones remotas de dos esclavos lleguen al maestro espaciadas por igual o menos que el tiempo que dura el timbre.

XII. EL MÓDULO ESCLAVO

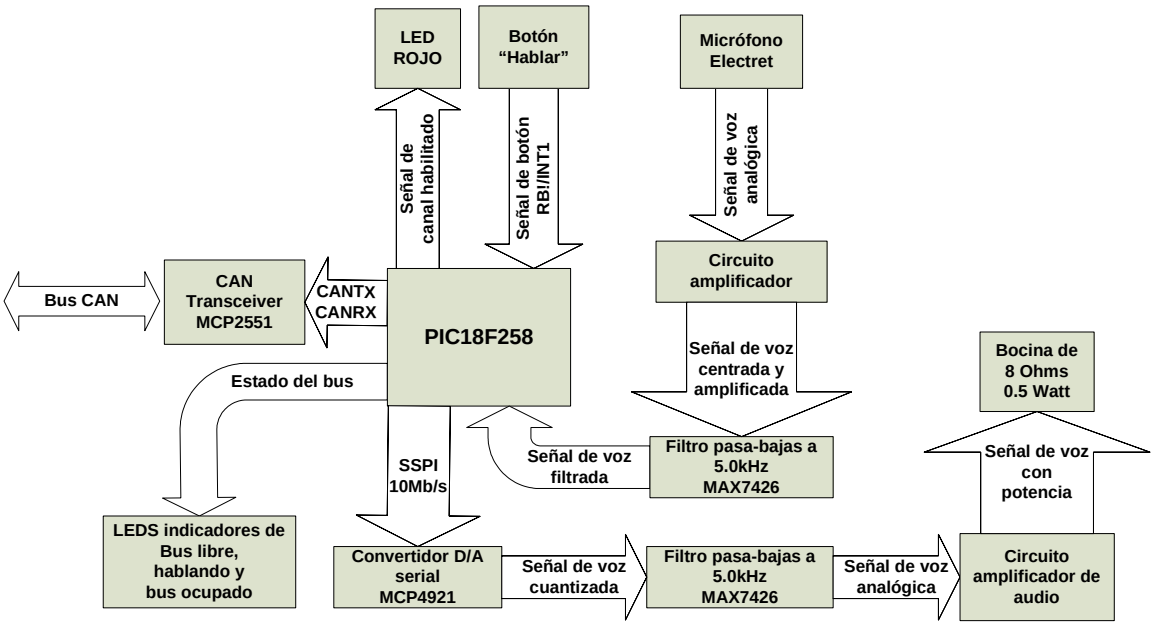


Figura 7: Diagrama de las interconexiones entre el PIC18F258 , sus periféricos y el bus CAN

La asignación de pines y módulos para el PIC15F258 está detallada en la Tabla 2**.** La Figura 13 proporciona el diagrama del circuito para el esclavo junto a la fuente de poder. El microcontrolador escogido para el módulo esclavo tiene las siguientes funciones a su cargo:

1. Determinar por medio de flancos el estado de la señal de botón y así decidir si en estado de envío de paquetes de voz o recepción mensajes de voz o remotos.

2. Utilizar su convertidor A/D para digitalizar la señal de voz, centrada en 2.5 V, amplificada por un circuito de transistores entre 0 y 5 V y además prefiltrada.
3. Enviar mensajes remotos a través del módulo CAN para solicitar habilitación del canal, i.e. permiso para enviar paquetes de voz.
4. Recibir mensajes empaquetados de voz y descomprimirlos para luego enviarlos a un convertidor D/A serialmente, después ser post-filtrados y enviados en forma de señal de voz reconstruida a un amplificador de potencia que alimenta una bocina.

5. Mostrar con tres LEDS el estado del bus (libre u ocupado) y del botón de hablar (presionado o no).

El diagrama de flujo del programa principal para el microcontrolador PIC18F258 está en la Figura 17. La atención de los interruptor para éste es igual a la del PIC18F458 en la Figura 15 y 16 excepto en el aspecto de que el esclavo no utiliza el Timer3.

A. Algoritmo de descompresión

Cuando un paquete de voz llega a un módulo esclavo o al maestro éste se almacena en un buffer que soporta la contención de hasta ocho mensajes completos. En la rutina principal del microcontrolador, si el botón de hablar no está presionado, la función asignada es continuamente chequear si el buffer de mensajes CAN tiene mensajes pendientes de lectura. Si ese es el caso el paquete de voz que contiene 8 bytes de datos se carga en un vector de recepción primero y si la descompresión anterior ha terminado, los datos se copian en un segundo vector de descompresión. La necesidad de usar este vector se verá a continuación.

Como el primer byte es siempre una la muestra completa del convertidor A/D, ésta se coloca directamente en el puerto serial SPI (Serial Peripheral Interface) que se conecta al convertidor D/A local (integrado MCP4921). Este proceso toma aproximadamente 4us de instrucciones y 4us del tiempo de colocación en el puerto del integrado.

Las siguientes 56 muestras son de a bit y se descomprimen de la misma forma en que el modulo transmisor tuvo que reconstruir la señal para dar tamaño al paso delta a utilizar. Es decir que cada bit se compara con sus cuatro predecesores y si son todos iguales el paso se duplica por etapas hasta un máximo de 32 de 255 bits o bien se disminuye directamente a la unidad.

El paso sumado o restado del registro de referencia donde se colocó la primera muestra aporta la magnitud de la muestra de la señal reconstruida que debe enviarse al convertidor D/A serialmente.

Luego de descomprimir cada bit, se chequea la recepción de otro paquete de voz, así, cuando se termina con el presente se dispone del nuevo en el vector de recepción. Una descripción más detallada del algoritmo de descompresión la provee la Figura 19.

B. Envío de mensaje remoto o requisición remota

No es posible para un módulo esclavo enviar una requisición remota en todo instante y esto tiene sentido por lo siguiente. Un esclavo puede enviar requisiciones remotas cuando el bus está ocupado ya sea por el maestro u otro esclavo, pero no cuando está recibiendo paquetes de voz del maestro ni cuando el canal está habilitado para él mismo y él se encuentre enviando paquetes de voz. Con esto queda cubierto el requisito de que el maestro se entere de que el esclavo quiere hablar con él sino es que ya lo está haciendo.

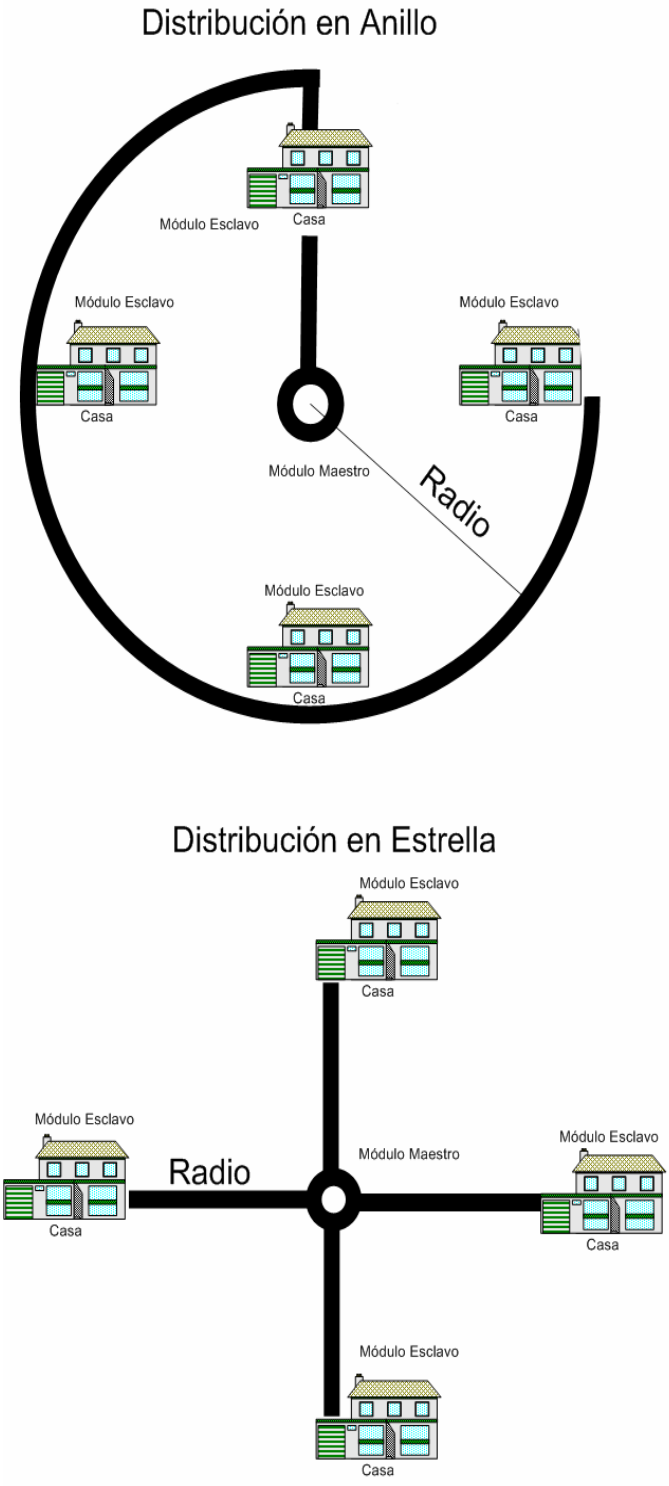
C. Amplificación de potencia en la señal de voz

Esta amplificación se refiere al módulo esclavo que no dispone de de un circuito analógico previo. Dicha amplificación se alcanza utilizando un transistor tipo Darlington (dos transistores cascadeados) con la capacidad de entregar hasta un Watt de potencia. Este energía garantiza el correcto funcionamiento de la bocina de 0.5 Watts propuesta en el circuito esclavo. La configuración utilizada puede apreciarse en el Diagrama del Circuito Esclavo.

D. Fuente de poder

El módulo esclavo necesita de una fuente de poder local para su funcionamiento. Esta fuente está proveída por medio de un transformador, un puente de diodos, capacitares, y dos reguladores de 5 voltios. Uno de los reguladores lo conforma un circuito integrado y provee la alimentación de todos los componentes de 5 voltios. El segundo regulador está construido con un diodo zener dos resistencias, y un transistor y su función es la de proveer el voltaje de referencia para los convertidores A/D y D/A. El circuito de la fuente está en la parte inferior del diagrama del circuito del módulo esclavo Figura 13.

XIII. CABLEADO DE LA RED



Una sencillo análisis de la distribución de señales en forma de anillo versus estrella demuestra cómo un anillo utiliza menos cableado que una estrella. A continuación se plantean dos de las más probables geometrías encontradas en la vida real que pueden verse desde el punto de vista del anillo o de la estrella. Son éstas por ejemplo la distribución en círculo (e.g. un hospital) y la línea recta (condominios u edificios de varios niveles). Para un círculo ambas distribuciones lucen como lo muestra la figura: Si se supone ***p*** el precio por unidad de longitud (metro) del cable par entorchado, ***r*** el radio del círculo y ***N*** el número de usuarios de la red de intercomunicadores, entonces las fórmulas para el Costo total del Anillo en Circulo (CAC_N) y el costo por unidad (CAC_n) se encuentran en la siguiente página ya que para demostrar que el anillo es más barato que la estrella, es necesario calcular primero la configuración en estrella. Por otro lado para el costo del círculo en configuración estrella (CEC) se tiene que el valor por unidad es $CEC_n = r \cdot p$ y el total es

$$CEC_N = \sum_{n=1}^{N} CEC_n = N \cdot r \cdot p$$

Figura 8: Diagrama de topología en círculo (anillo y estrella)

El costo total del anillo en círculo y del mismo por unidad es respectivamente:

$$CAC_N = \left(2 \cdot \pi \cdot r - \left(\frac{2 \cdot \pi \cdot r}{N}\right) + r\right) \cdot p$$

$$= \left[\left(\frac{N-1}{N}\right) \cdot 2 \cdot \pi + 1\right] \cdot r \cdot p$$

$$CAC_n = \left(\left(\frac{N-1}{N}\right) \cdot 2 \cdot \pi + 1\right) \cdot \frac{1}{N} \cdot r \cdot p$$

Las fórmulas anteriores aprovechan el hecho de que no es necesario cerrar el círculo para completar el anillo lógico. De allí que se substraiga el segmento de curva $\frac{2 \cdot \pi \cdot r}{N}$ del perímetro, asumiendo que los usuarios se encuentran uniformemente distribuidos alrededor del círculo a distancias iguales a la del segmento de curva.

Como el costo total para la configuración estrella y anillo se diferencian únicamente en el término N y el término $\left[\left(\frac{N-1}{N}\right) \cdot 2 \cdot \pi + 1\right]$, se desea saber que valores de N hacen menos costoso el anillo que la estrella resolviendo la siguiente inecuación:

$$\left[\left(\frac{N-1}{N}\right) \cdot 2 \cdot \pi + 1\right] \cdot \frac{1}{N} < N$$

$$\left[\left(\frac{N-1}{N}\right) \cdot 2 \cdot \pi + 1\right] < N^2$$

$$(N-1) \cdot 2\pi + N < N^3$$

$$N^3 - (2 \cdot \pi + 1) \cdot N + 2\pi > 0$$

Resulta necesario resolver la inecuación encontrando sus raíces al volverla una ecuación:

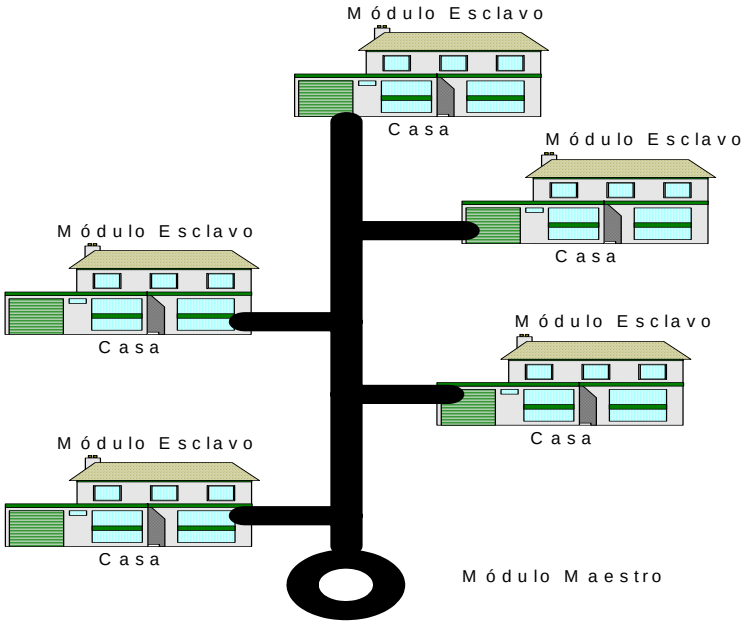
$$N^3 - (2 \cdot \pi + 1) \cdot N + 2\pi = 0$$

$$N_1 = 2.06 \quad , \quad N_2 = 1 \quad , \quad N_3 = -3.06$$

$$N > 2.06$$

Las raíces son los límites de los intervalos en los que se evalúa para resolver la inecuación. Se obtiene que la inecuación se cumple si N<-.3.06 , 0 < N <1 y N>2.06. Por lo tanto, descartando un numero de usuarios menor a la unidad, el anillo resulta más barato que la estrella si la cantidad de usuarios es estrictamente mayor a las 2.06 unidades, es decir al menos 3 usuarios.

Distribución en Anillo



Distribución en Estrella

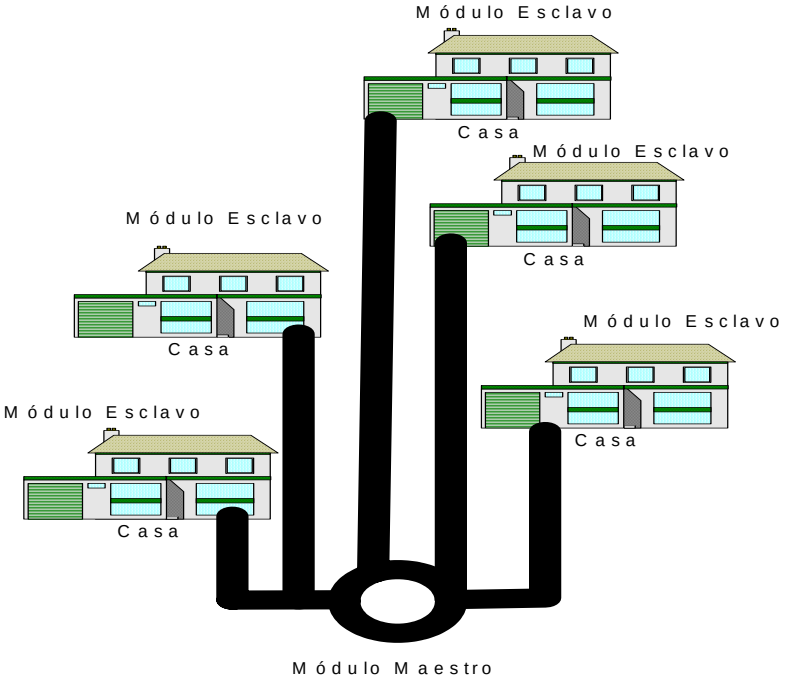


Figura 9: Diagrama de Topología en Línea Recta (Anillo y Estrella)

En cuanto a cablear en línea recta, esto es más común en un edificio porque teniendo un módulo maestro centralizado, los usuarios tienden a ubicarse progresivamente más lejos del mismo. La distribución luciría como en la figura adyacente. El cálculo del costo total para la infraestructura de una red en anillo en línea recta con usuarios alejados a múltiplos de una unidad constante diferente a uno del maestro se calcula así:

Si D es la distancia del usuario más alejado y k la distancia de cada usuario al bus, entonces el costo total del anillo en recta (CAR) sería:

$$CAR_N = [D + (k \cdot N)] \cdot p$$

y el costo por usuario:

$$CAR_n = \frac{[D + (k \cdot N)] \cdot p}{N} = \left(\frac{D}{N} + k\right) \cdot p$$

Para la distribución estrella sobre una recta (CER) el costo por unidad es:

$$CER_n = \left[\left(\frac{D}{N}\right) \cdot n + k \cdot\right] p$$

El costo total sería $CER_N = \sum_{n=1}^{N} \left(\frac{D}{N} \cdot n + k\right) \cdot p$

Es evidente entonces que el costo por usuario es menor para el anillo lógico sobre una recta.

XIV. RESULTADOS

El resultado principal de este proyecto es la red de intercomunicadores funcional. Se compone de 10 módulos esclavos y uno maestro interconectados por el protocolo CAN funcionando a 125kbits/seg. El módulo maestro y esclavo comprimen datos de voz muestreándolos a 25kHz. El maestro es efectivamente capaz de detectar mensajes remotos en plena compresión o descompresión de paquetes de voz. La distancia corroborada de funcionamiento del bus es de 400m. A partir de 450m los módulos ya no perciben el estado recesivo del bus de manera estable.

Dado que los microcontroladores de la familia PIC18FXX8 no pueden darse el lujo de muestrear la señal de voz a 32kHz por limitaciones de velocidad y dado que no es ésta su única tarea durante le etapa de muestreo, resultó ser necesario muestrear a frecuencias inferiores a la comúnmente utilizada en un algoritmo ADPCM. Sin embargo,

A partir de la simulación que se hizo para el muestreo de voz, se llegó a la conclusión de que una frecuencia de muestreo entre 16kHz y 32kHz también arroja resultados aceptables. Esto lo muestran las Gráficas 7 a 11, las cuales se tomaron con una frecuencia de muestreo de 23kHz, la cual resultó siendo la utilizada en el proyecto. En ellas puede notarse como con la disminución de la frecuencia, la señal de salida con respecto a la de entrada se degrada. Sin embargo, la señal de voz obtenida en cualquiera de los dos módulos es clara e inteligible y esto gracias al pre- y post-filtrado además de que la frecuencia de muestreo no es tan baja como los 16 kHz que aún se podrían considerar viables.

En cuanto al precio de un proyecto como el presentado, el costo de los componentes para la de la parte digital de un módulo maestro es de alrededor de los 20 dólares. En cuanto al módulo esclavo, el precio de sus dispositivos asciende a los 30 dólares aproximadamente.

Los precios fueron cotizados del catálogo de “JAMECO Electronics”, así como de la página de ”MAXIM Integrated Products”. El desglose de cada componente se encuentra en las Tablas 3 y 4.

La utilización del integrado MAXIM7426 constituyó un ahorro en cuanto a que funciona con un voltaje de 5 voltios y tierra como referencia. De no haberse aprovechado este integrado, los filtros se habrían construido con amplificadores operacionales que requieren además de un voltaje mayor a los 5 Voltios (debido a la aplicación) también de polaridad negativa. Esto significa un ahorro en el transformador que puede ser de menores dimensiones.

Otro dispositivo que por defecto requería de voltaje negativo era el convertidor análogo digital que en un principio se pensó utilizar, el DAC0808. Sin embargo éste terminó siendo sustituido por el MCP4921 que al recibir sus datos serialmente no sólo ahorra puertos en el microcontrolador, sino que funciona con 5 voltios y tierra.

XV. CONCLUSIONES

1. Se determinó que la variante del método Delta Adaptativa más apropiado para la aplicación era la de muestrear sobre los 16 kHz y cuantificar a un bit 56 de las 57 muestras a enviar en un paquete de voz CAN y la primera muestra cuantificarla a 8 bits.

2. Se simuló la aplicación del protocolo Delta Adaptativa sobre una señal de voz., variando sus parámetros para analizar ventajas y desventajas.

3. Se construyó una red CAN para un panel de control con al menos 10 intercomunicadores individuales o esclavos.

4. La ocurrencia de colisiones de las señales en un sistema de intercomunicadores para diez intercomunicadores individuales en red, contemplando señales de control y señales de voz, resulta remota y si se da es supervisada automáticamente por el módulo CAN de los microcontroladores.

5. El protocolo de comunicación CAN sumado al arbitraje en software permiten el transporte de la máxima cantidad de datos de voz comprimida, sin que se pierdan datos de control como activación de timbres o alarmas.

6. Se adaptó la red digital a los intercomunicadores y al panel central a existentes introduciendo como nuevo componente el PIC18F258 en los intercomunicadores individuales o esclavos y el PIC18F458 en el panel central, haciendo uso de la fuente de poder y señales de voz y direccionamiento integrados en el dispositivo analógico .

7. Se digitalizaron las señales analógicas utilizando filtros y el módulo convertidor análogo/ digital del microcontrolador. La conversión digital/analógica por medio de dispositivos DAC.

8. La lógica que permite multiplexar los canales de voz en el módulo central para distribuirlas según convenga al destinatario quedó dada por la asignación de identificadores de alta prioridad para las requisiciones remotas y las inmediatamente superiores a estas para enviar paquetes de voz desde y hacia el módulo maestro.

9. Se determinó que durante la transmisión de paquetes de voz el bus está ocupado en un 36% lo que facilita el envío de mensajes remotos.

10. Se establecieron prioridades de identificador y de envío para el procesamiento de cada señal según su tipo (voz, o control) y origen.

11. Los límites superiores de distancia y velocidad del bus CAN soportados en la instalación de intercomunicadores es de al menos 400m y como máximo la longitud especificada teóricamente de 500m para la velocidad de 125kbits/seg.

12. Los costos para la construcción de cada módulo digital superan a los de los analógicos, pero la reducción de cables trae consigo un ahorro sustancial, según se vió en las configuraciones anillo y estrella.

13. La red de intercomunicadores construida resulta ser una posibilidad viable para utilizar. En cuanto a su comercialización ésta resulta un poco difícil especialmente por el elevado costo del módulo esclavo.

XVI. RECOMENDACIONES

Es necesario señalar que la implementación de un bus digital moderno como el CAN abre la posibilidad a la realización de muchas aplicaciones sobre este mismo proyecto. Sin embargo el planteamiento que se hace para el mismo no explota todas estas posibilidades debido a que se busca en primera instancia concretar de forma realista lo que se propone.

Debido a lo anterior en este proyecto se renuncia a habilitar la posibilidad de conversaciones alternantes entre módulos individuales de intercomunicador, esto se deja como una posibilidad en el futuro. Otras ampliaciones al proyecto podrían ser el hecho de convertir la red en un medio de transmisión con capacidad full-duplex simulada por multiplexación en tiempo. Esto requeriría de encontrar la forma de aumentar la velocidad en el bus y aplicar un nuevo y más eficiente algoritmo de compresión (aplicable al mismo microcontrolador o sino utilizando un microcontrolador más avanzado) sin olvidar conservar las funciones básicas del sistema y eliminar el problema que presenta la retroalimentación.

Para que la red de intercomunicadores cubra mayores distancias es necesario reducir la velocidad del bus CAN y por lo tanto utilizar un aún más eficiente algoritmo de compresión de voz, es decir algún Vocoder (Voice Coder) como el CELP(Code Excited Linear Prediction) o el VSELP (Vector Sum Excited Linear Prediction). Estos reducen el ancho de banda para la transmisión de voz hasta alrededor de los 4.0 kbits/seg. Sin embargo requerirían de Microcontroladores más rápidos y enfocados al procesamiento de Señales digitales debido a la carga analítica y de procesamiento que conllevan dichos algoritmos.

La red construida en este proyecto sin embargo podría ser utilizable en edificios o condominios, etc. donde las distancias superiores a los 400m en el recorrido del bus CAN no sean superadas.

Se sugiere construir un monitor del bus CAN para determinar en una computadora (por puerto serial o USB por ejemplo) los mensajes que viajan en el mismo y su procedencia en cualquier instante dado (datos digitales de voz o señales de control). Así se podrían grabar conversaciones y registrar su ocurrencia en el tiempo.

Otra opción viable para incrementar la distancia efectiva de funcionamiento del sistema es utilizar un módulo extensor del bus CAN que convierte un par de cobre en un par de fibra óptica.

XVII. BIBLIOGRAFÍA

Marven, Craig ; Gillian Ewers. 1996. *A Simple Aproach to Digital Signal Processing.* Topics in Digital Signal Processing. E.U.A. John Wiley & Sons, Inc. 236 págs.

Palarea, Gonzalo. 1995. *Efectos de la Variación de la Frecuencia de Muestreo y Bits de Cuantificación en la Digitalización de Señales Analógicas*. Guatemala. 124 págs.

Richards, Path . Microchip Technoloy Inc. 2002. *Application Note 228, A CAN Physical* Layer Discussion. E.U.A. 22 págs.

PIC18Fxx8 Data Sheet. 2003. Microchip Technology Inc. E.U.A. 400 págs.

PIC18Cxxx Reference Manual. 2000. Microchip Technology Inc. *Section 22.CAN* . E.U.A. Págs. 22-1 a la 22-80.

MCP2551 Data Sheet. 2003. Microchip Technology Inc. High Speed CAN Transceiver E.U.A. 19 págs.

MCP4921 Data Sheet. 2004. Microchip Technology Inc. 12-Bit DAC with SPI™ Interface. E.U.A. 27 págs.

MAX7426/7427. 2000.Maxim Integrated Products. 5TH Order, Lowpass, Elliptic, Switched-Capacitor Filters. E.U.A. 12 págs.

MPLAB®C18 COMPILER GETTING STARTED. 2003. Microchip Technology Inc. E.U.A. 56 págs.

MPLAB® C18 COMPILER USER'S GUIDE. 2003. Microchip Technology Inc. E.U.A. 104 págs.

MPLAB® C18 COMPILER LIBRARIES. 2003. Microchip Technology Inc. E.U.A. 152 págs.

MICROCHIP APPLICATION MAESTRO™ USER'S GUIDE. 2003. Microchip Technology Inc. E.U.A. 28 págs.

MICROCHIP Interrupt based CAN Library Module. 2003. Microchip Technology Inc. E.U.A. 8 págs.

SN54/74LS147, SN54/74LS148. 2003. MOTOROLA. 10-LINE-TO-4-LINE AND 8-LINE-TO-3-LINE PRIORITY ENCODERS. E.U.A. 5 págs.

DM74LS154. 2000. FAIRCHILD SEMICONDUCTOR™. 4-Line to 16 Line Decoder/DeMultiplexer. E.U.A. 5 págs.

XVIII. APÉNDICE

TABLAS

Tabla 2: Designación de los Pines en el Microcontrolador PIC18F458

Nombre	Función	Descripción
RA0	AN0	Entrada analógica 0 Convertidor A/D (SG-IN)
RA1	I/O	Entrada/Salida digital sin uso
RA2	Vref+	Referencia positiva convertidor A/D
RA3	Vref-	Referencia negativa convertidor A/D
RA4	I/O	Entrada/Salida digital sin uso
RA5	SS	Salida digital CHIP SELECT hacia convertidor D/A
RB0	I/O	Entrada/Salida digital sin uso
RB1	INT1	Entrada detectora de flancos push button remoto (TOGG)
RB2	CANTX	Salida módulo CAN
RB3	CANRX	Entrada módulo CAN
RB4	Output	Salida digital dirección timbre T0
RB5	Output	Salida digital dirección timbre T1
RB6	Output	Salida digital dirección timbre T2
RB7	Output	Salida digital dirección timbre T3
RC0	I/O	Entrada/Salida digital sin uso
RC1	I/O	Entrada/Salida digital sin uso
RC2	I/O	Entrada/Salida digital sin uso
RC3	SCK	Salida reloj módulo SPI hacia Convertidor D/A
RC4	SDI	Entrada datos módulo SPI (sin uso a GND)
RC5	SDO	Salida datos módulo SPI hacia convertidor D/A
RC6	Output	Salida digital hacia activación/desactivación pre-filtro
RC7	Output	Salida digital hacia activación/desactivación post-filtro
RD0	Input	Entrada digital dirección modulo CAN DIR0
RD1	Input	Entrada digital dirección módulo CAN DIR1
RD2	Input	Entrada digital dirección módulo CAN DIR2
RD3	Input	Entrada digital dirección módulo CAN DIR3
RD4	Input	Entrada digital dirección módulo CAN DIR4
RD5	Input	Entrada digital dirección módulo CAN DIR5
RD6	Input	Entrada digital dirección módulo CAN (a GND)
RD7	Input	Entrada digital dirección módulo CAN (a GND)
RE0	Output	Salida digital hacia LED indicador BUS FREE (verde)
RE1	Output	Salida digital hacia LED indicador TALKING (amarillo)
RE2	Output	Salida digital hacia LED indicador BUS BUSY(rojo)
OSC1	Input	Entrada oscilador 10 MHz
OSC2	Output	Salida oscilador 10 MHz
MCLR	Entrada	Reset del microcontrolador a Vdd

Tabla 3: Designación de los Pines en el Microcontrolador PIC18F258

Nombre	Función	Descripción
RA0	AN0	Entrada analógica 0 convertidor A/D
RA1	I/O	Entrada/Salida digital sin uso
RA2	Vref+	Referencia positiva convertidor A/D
RA3	Vref-	Referencia negativa convertidor A/D
RA4	I/O	Entrada/Salida digital sin uso
RA5	SS	Salida digital CHIP SELECT hacia convertidor D/A
RB0	I/O	Entrada/Salida digital sin uso
RB1	INT1	Entrada detectora de flancos push-button local
RB2	CANTX	Salida módulo CAN
RB3	CANRX	Entrada módulo CAN
RB4	I/O	Entrada/Salida digital sin uso
RB5	Output	Salida digital hacia LED indicador BUS FREE (verde)
RB6	Output	Salida digital hacia LED indicador TALKING (amarillo)
RB7	Output	Salida digital hacia LED indicador BUS BUSY(rojo)
RC0	I/O	Entrada/Salida digital sin uso
RC1	I/O	Entrada/Salida digital sin uso
RC2	Output	Salida digital indica match de dirección CAN
RC3	SCK	Salida reloj módulo SPI hacia convertidor D/A
RC4	SDI	Entrada datos módulo SPI (a GND)
RC5	SDO	Salida datos módulo SPI hacia convertidor D/A
RC6	Output	Salida digital hacia activación/desactivación pre-filtro
RC7	Output	Salida digital hacia activación/desactivación post-filtro
OSC1	Input	Entrada oscilador 10 MHz
OSC2	Output	Salida oscilador 10 MHz
MCLR	Entrada	Reset del microcontrolador a Vdd

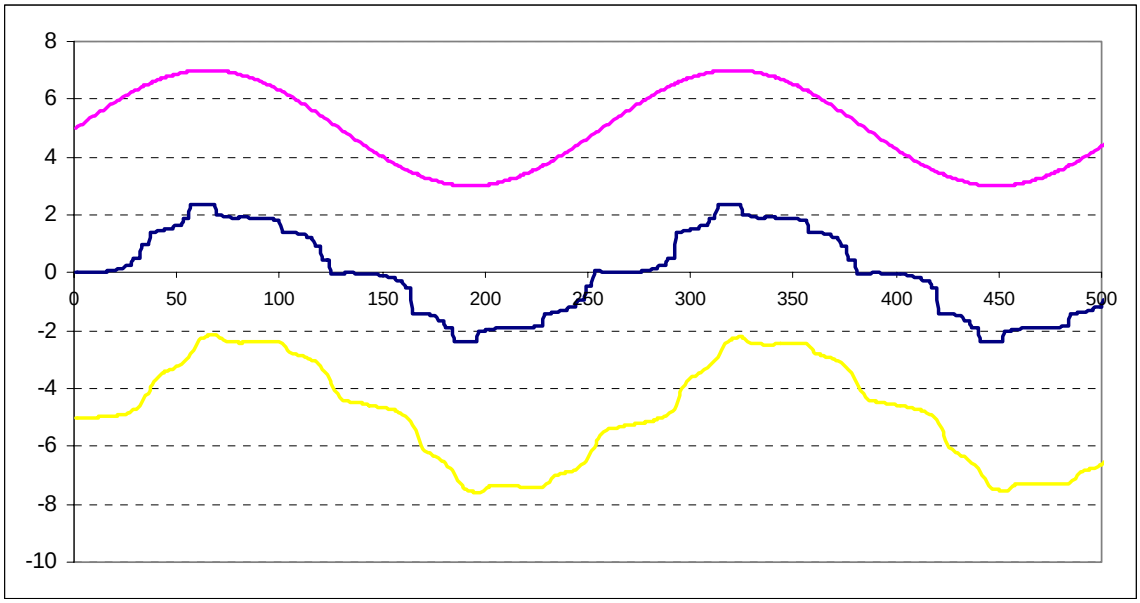
Tabla 4: Costos de los componentes electrónicos módulo maestro

Componente	Descripción	Cantidad	Precio
PIC18F458	Microcontrolador 10MIPS	1	$ 9.79
MCP4921	Convertidor D/A serial	1	$ 1.50
MCP2551	CAN Transceiver	1	$ 1.50
MAX7426	Filtro pasabajas elíptico / 5 polos	2	$ 4.00
74LS148	Codificador de 8 a 3 de prioridad	2	$ 1.60
74LS154	Decodificador de 4 a 16	1	$ 1.05
Amplificación	Transistores, capacitores y resistencias		$ 1.00
	Total	7	$ 20.44

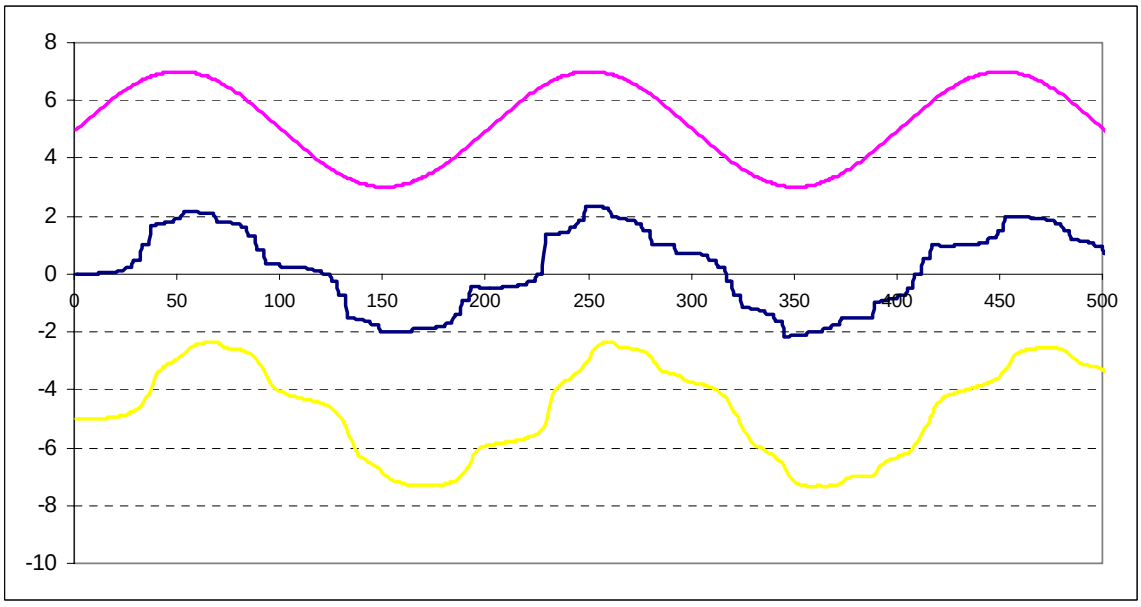
Tabla 5: Costos de los componentes electrónicos módulo esclavo

Componente	Descripción	Cantidad	Precio
PIC18F258	Microcontrolador 10MIPS	1	$ 9.65
MCP4921	Convertidor D/A serial	1	$ 1.50
MCP2551	CAN Transceiver	1	$ 1.50
MAX7426	Filtro pasabajas elíptico / 5 polos	2	$ 4.00
NTE 48	Darlington de alta corriente	1	$ 0.95
Ferrite Speaker	Bocina 8 Ohms 0.5 Watts	1	$ 2.00
Cartucho de Micrófono	Condensador Electret Omnidir.	1	$ 0.50
Fuente de Poder	Transformador 6.3V@ 0.50Amp	1	$ 4.00
Diodos para Puente	NTE116 1N4001	4	$ 2.00
Regulador 5V	NTE-960	1	$ 1.65
Amplificación	Transistores, capacitores y resistencias		$ 1.00
	Total	9	$28.75

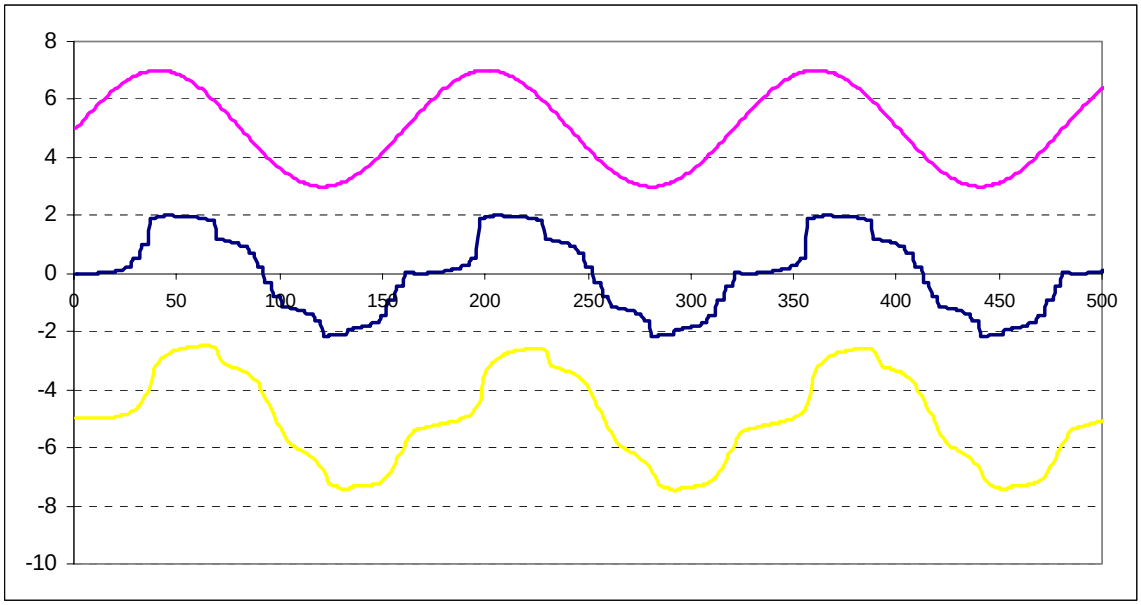
Gráfica 1: Función senoidal de 4 Vp-p a 500Hz muestreada a 32kHz con ADPCM adaptado a CAN y filtrado a 4kHz.



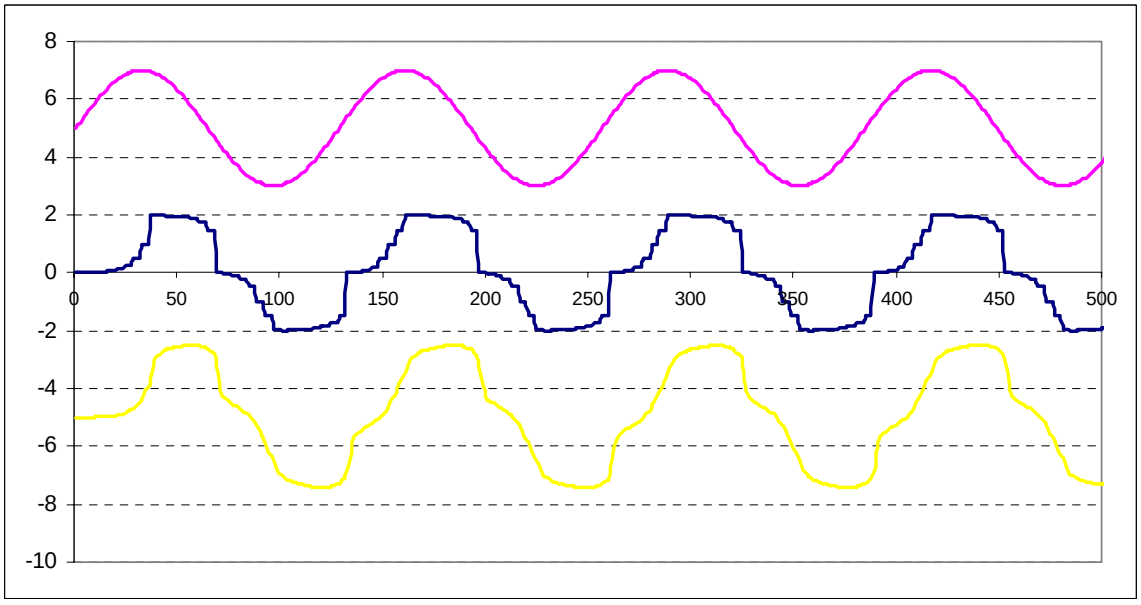
Gráfica 2: Función senoidal de 4 Vp-p a 500Hz muestreada a 25kHz con ADPCM adaptado a CAN y filtrado a 4kHz.



Gráfica 3: Función senoidal de 4Vp-p a 500Hz muestreada a 20 kHz con ADPCM adaptado a CAN y filtrado a 4kHz.

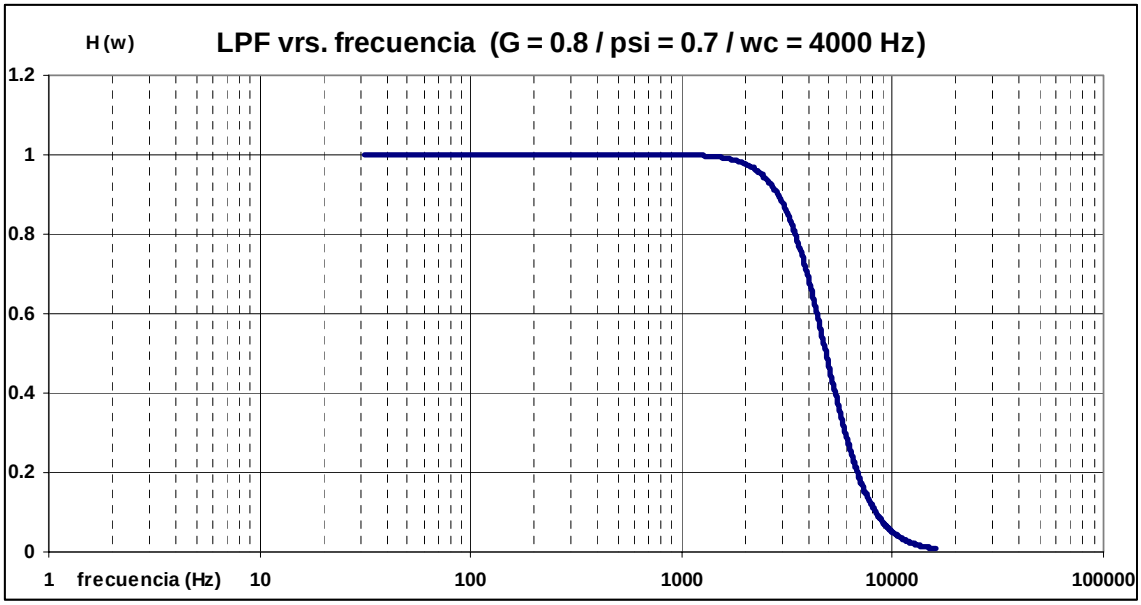


Gráfica 4: Función senoidal 4 Vp-p a 500Hz muestreada a 16 kHz con ADPCM adaptado a CAN y filtrado a 4kHz.

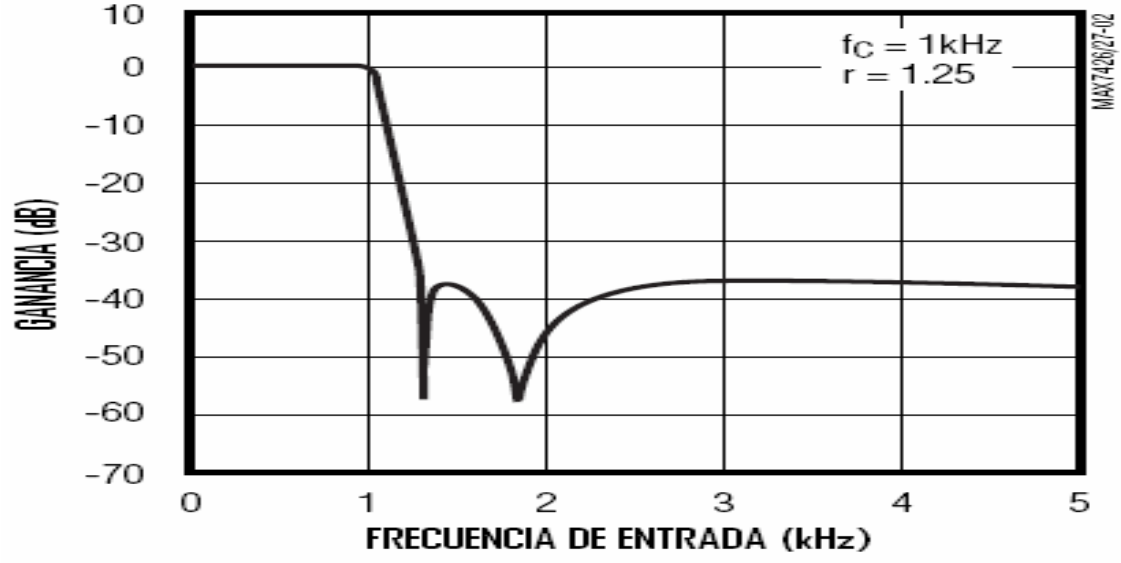


Grafica 5: Función de transferencia filtro del pasa-bajas tipo Butterworth de 4 polos simulado

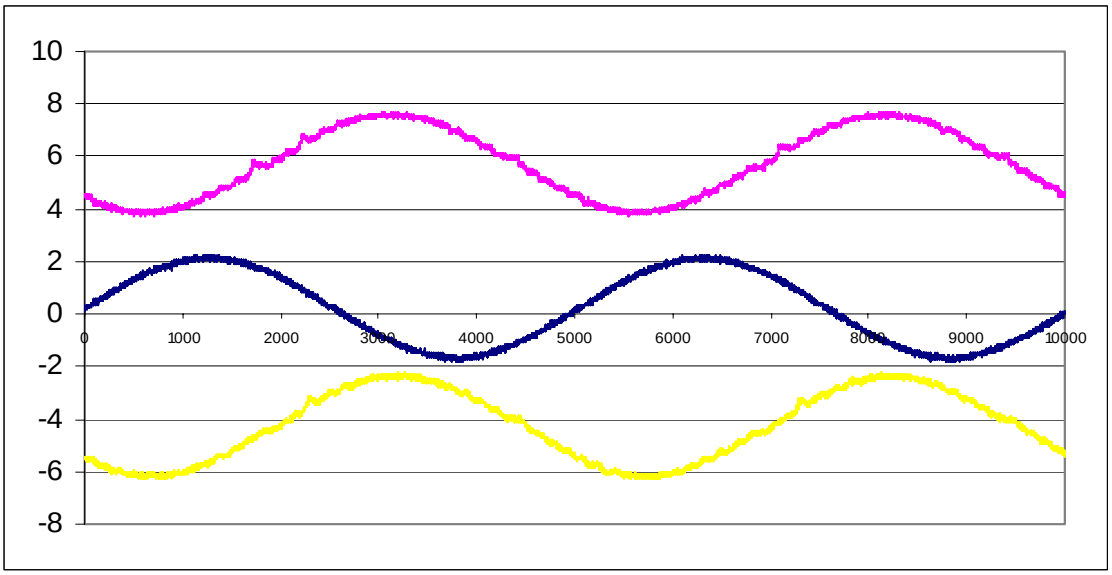
$$H(s)=\frac{\omega_c^4\cdot(\omega^4-2\cdot\omega_c^2\cdot(2\cdot\xi^2+1)\cdot\omega^2+\omega_c^4)}{(w_c^4+2\cdot(2\cdot\xi^2-1)\cdot\omega_c^2\cdot\omega^2+\omega^4)^2}+\frac{4\cdot\xi\cdot\omega_c^5\cdot\omega\cdot(\omega^2-\omega_c^2)}{(w_c^4+2\cdot(2\cdot\xi^2-1)\cdot\omega_c^2\cdot\omega^2+\omega^4)^2}\cdot i$$



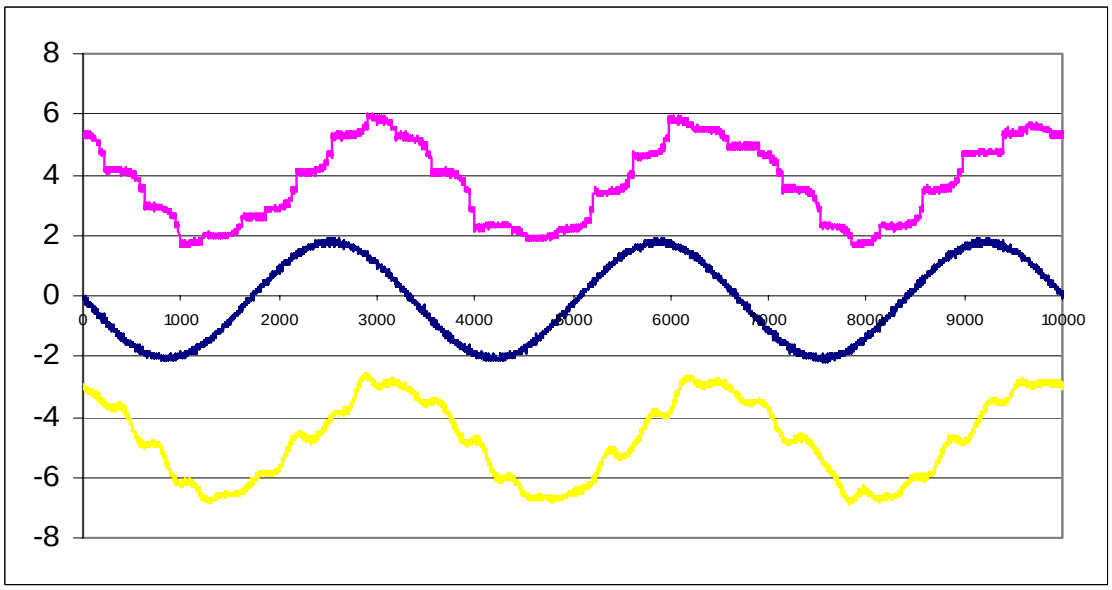
Grafica 6: Función de transferencia filtro del pasa-bajas elíptico de 5 polos implementado con el MAX7426



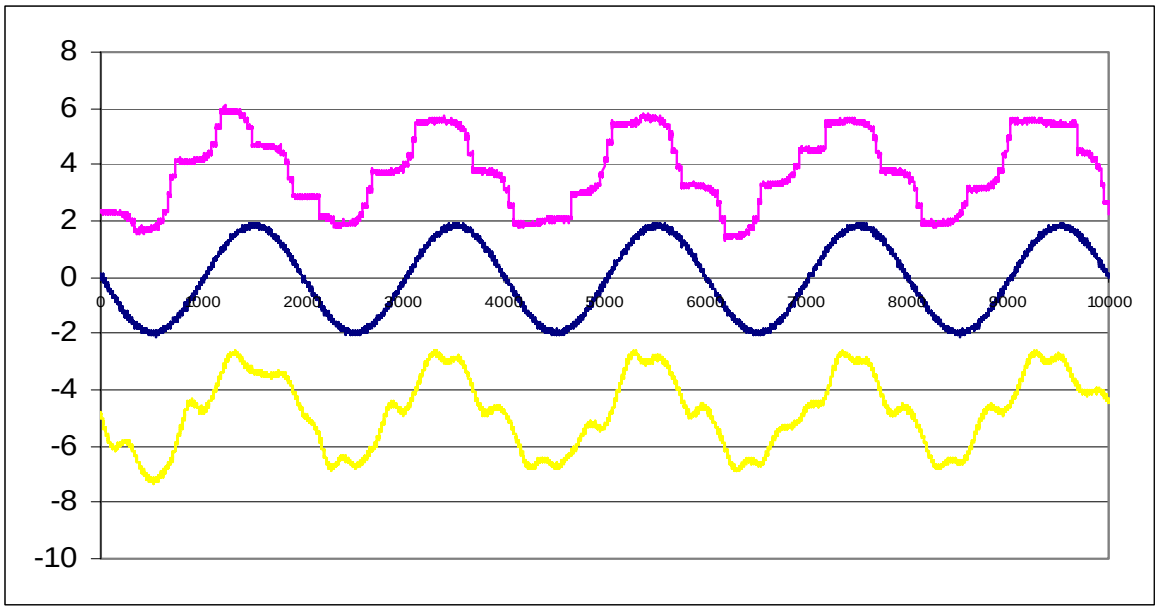
Gráfica 7: Función senoidal de 4 Vp-p a 100Hz muestreada 23 kHz con ADPCM adaptado a CAN (superior reconstruida a 23kHz, central referencia, e inferior filtrada).



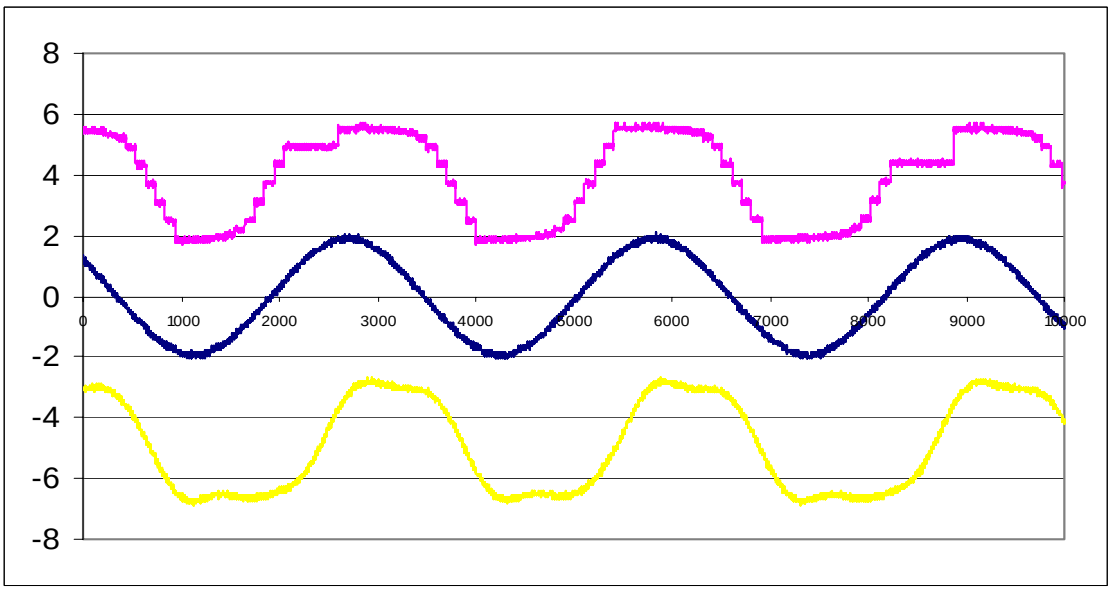
Gráfica 8: Función senoidal de 4 Vp-p a 300Hz muestreada a 23 kHz con ADPCM adaptado a CAN (superior reconstruida a 23kHz, central referencia, e inferior filtrada).



Gráfica 9: Función senoidal de 4 Vp-p a 500Hz muestreada a 23 kHz con ADPCM adaptado a CAN (superior reconstruida a 23kHz, central referencia, e inferior filtrada).



Gráfica 10: Función senoidal de 4 Vp-p a 800Hz muestreada a 23 kHz con ADPCM adaptado a CAN (superior reconstruida a 23kHz, central referencia, e inferior filtrada).



Gráfica 11: Función senoidal de 4 Vp-p a 1kHz muestreada a 23 kHz con ADPCM adaptado a CAN (superior reconstruida a 23kHz, central referencia, e inferior filtrada).

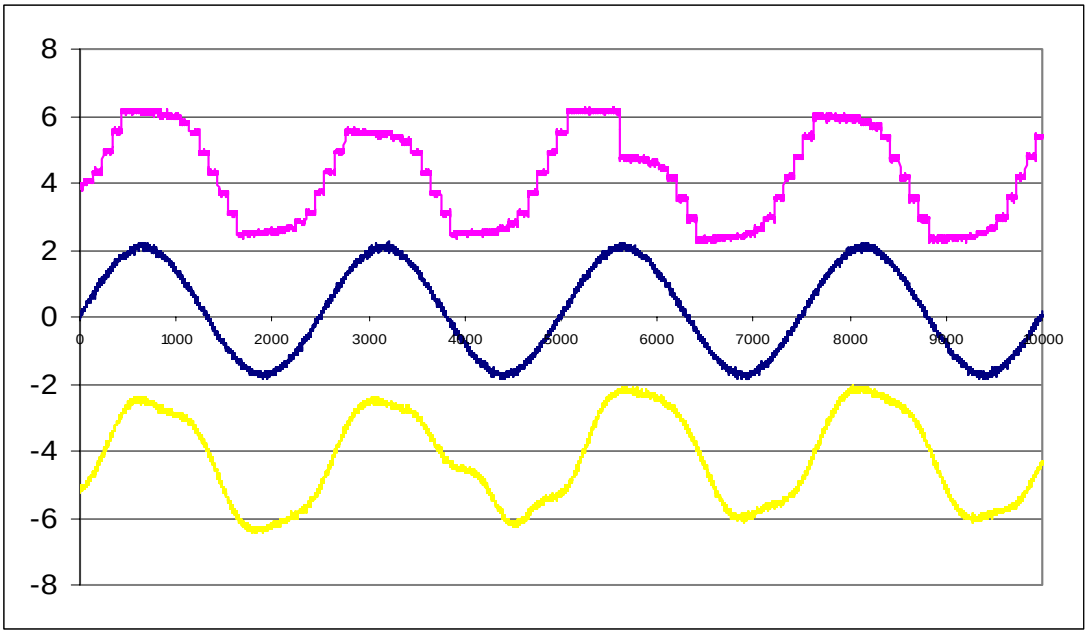


Figura 10: Dibujo de los módulos intercomunicadores maestro y esclavos en su versión no modificada con el bus CAN

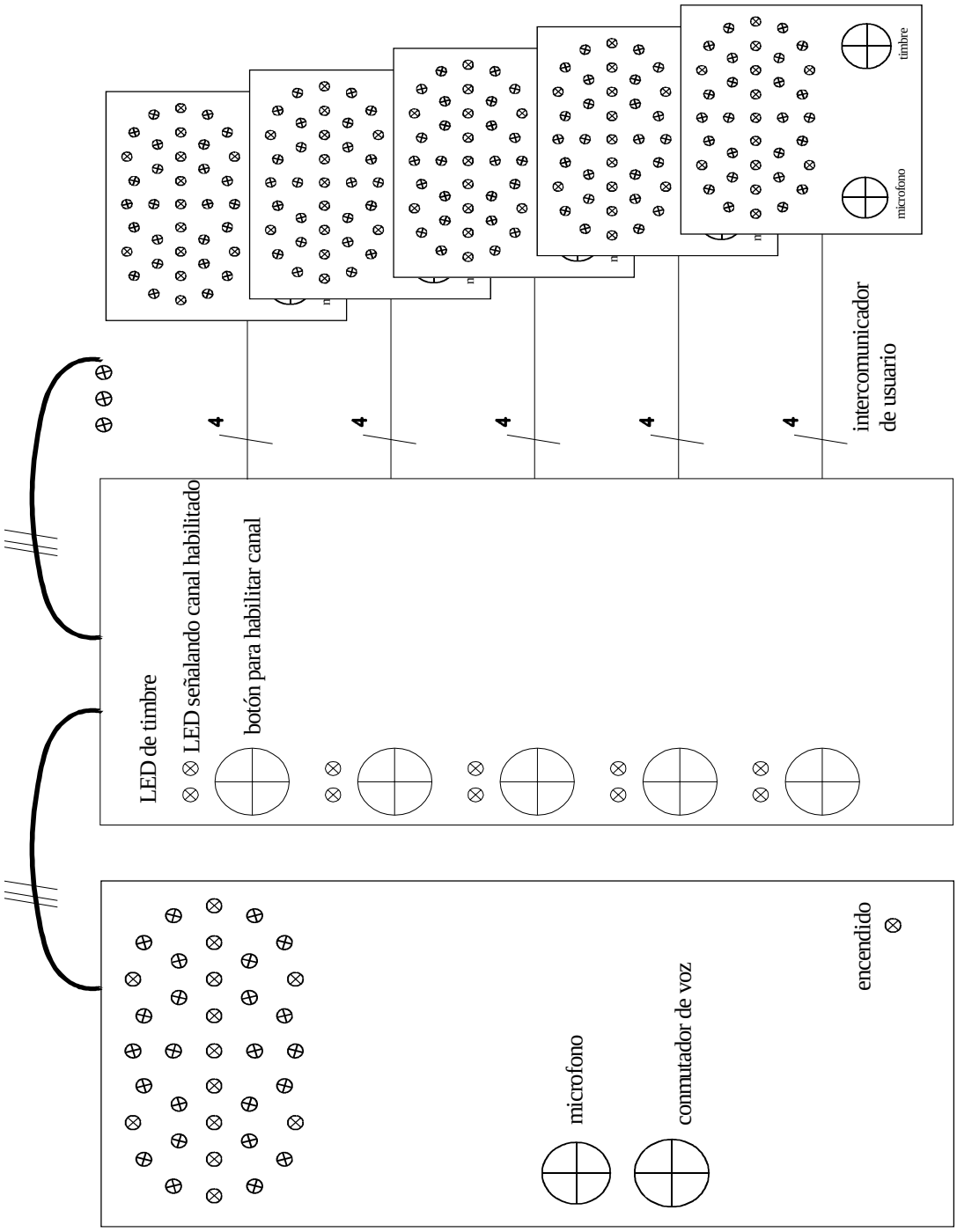


Figura 11: Circuito maestro parte 1

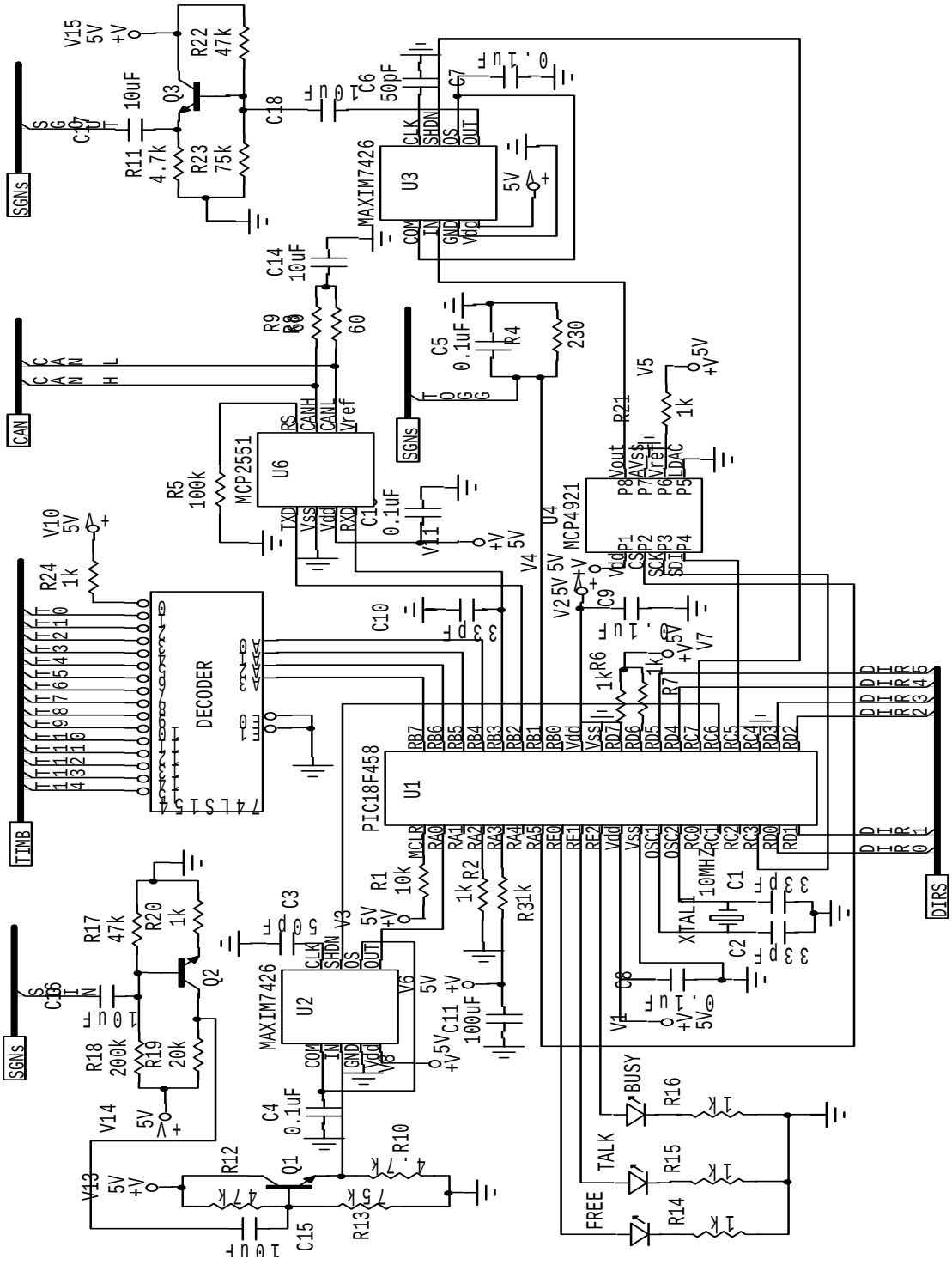


Figura 12: Circuito maestro parte 2

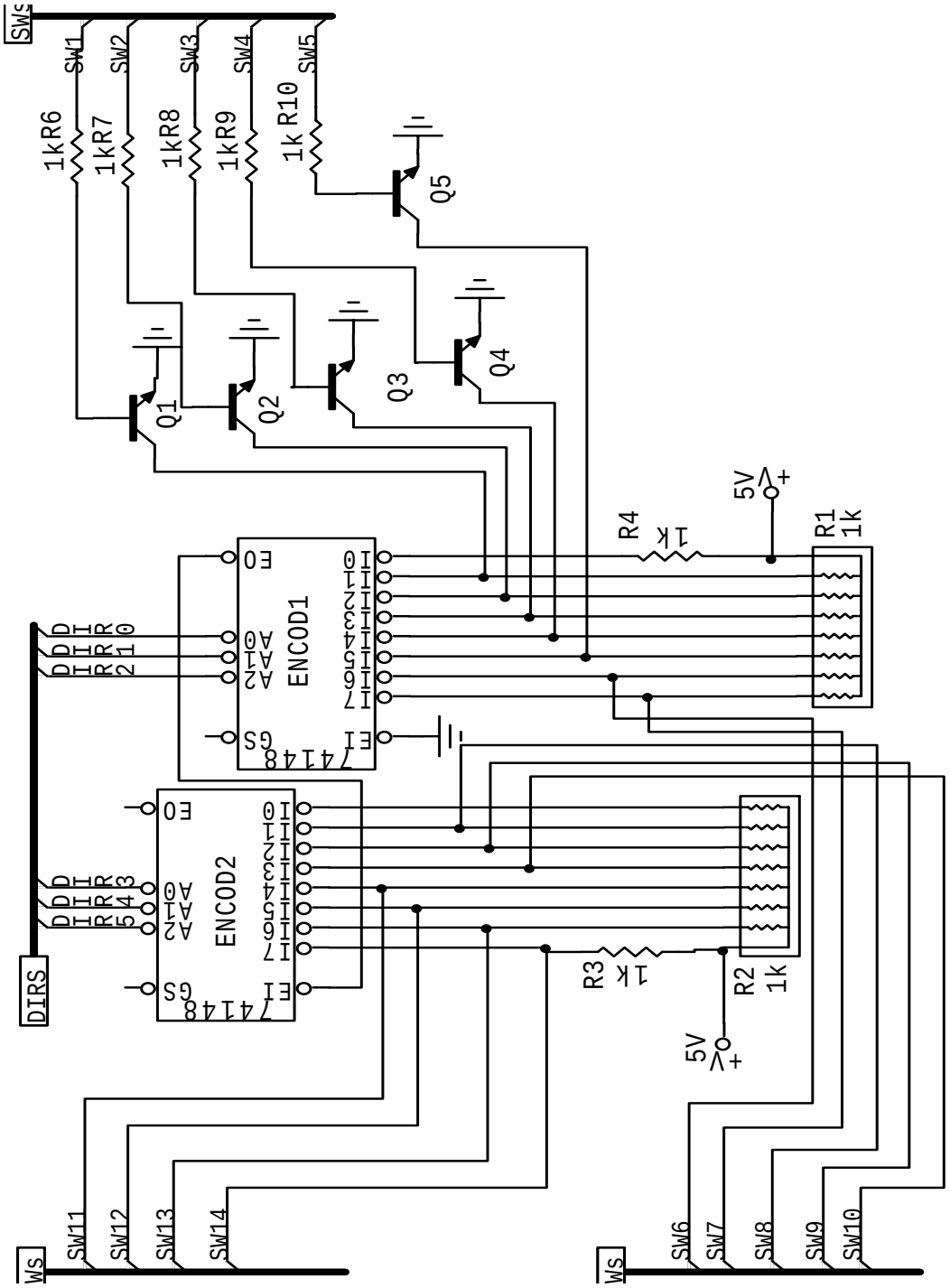


Figura 13: Circuito esclavo

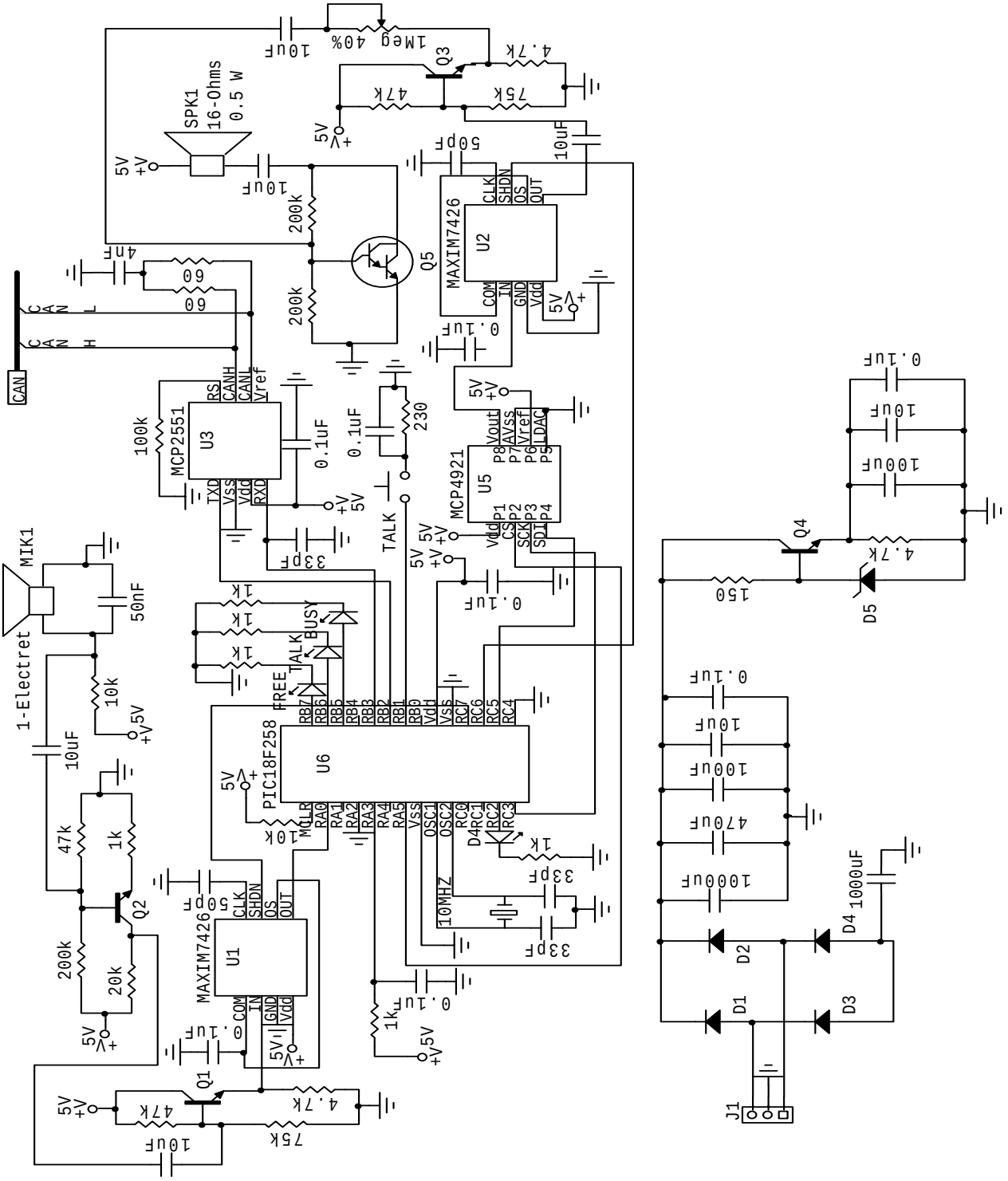


Figura 14: Diagrama de flujo del programa principal en el PIC18F458

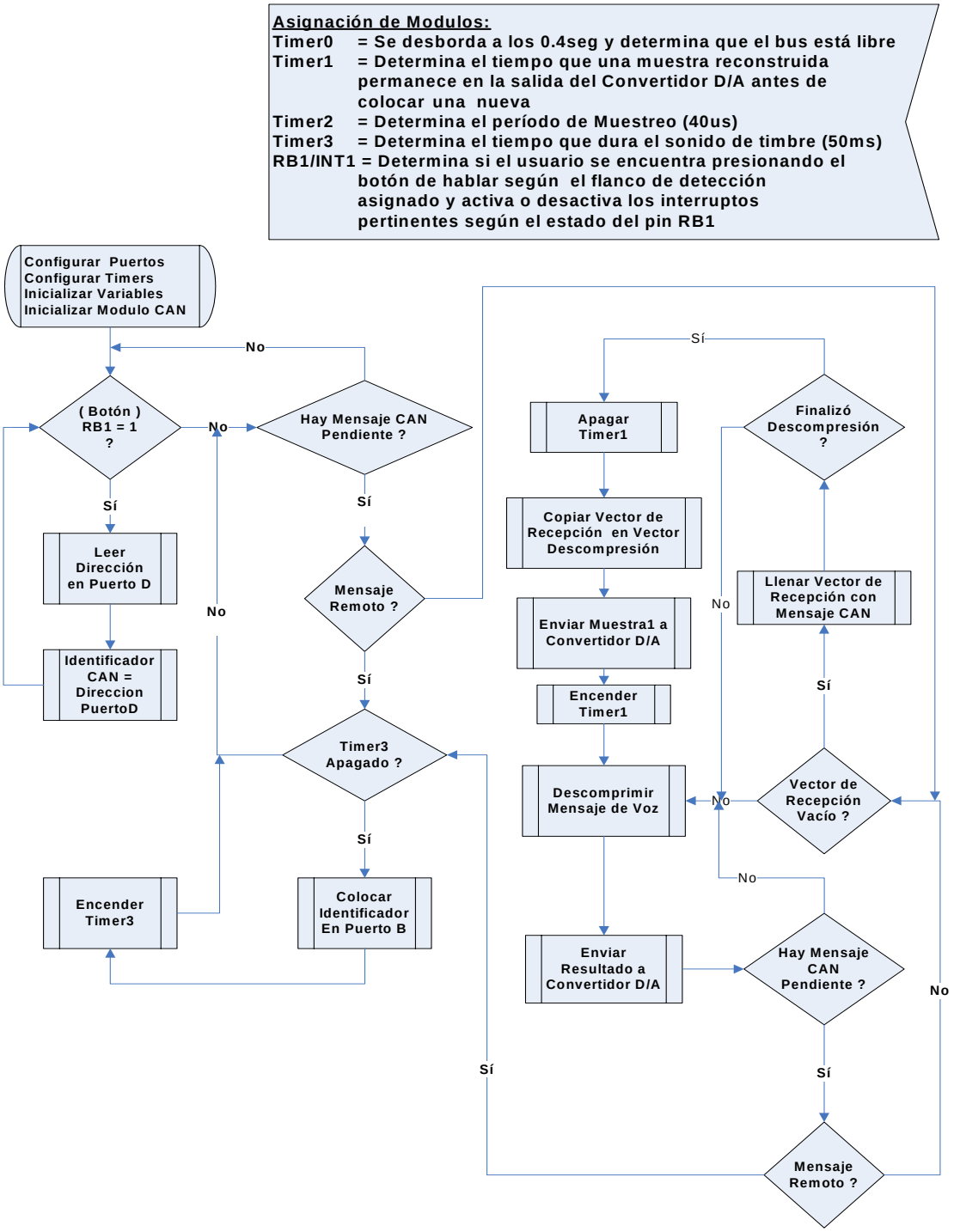


Figura 15: Diagrama de flujo de la secuencia de atención de interrupciones de alta prioridad en el microcontrolador PIC18F458

Figura 16: Diagrama de flujo de la atención de interrupciones de baja prioridad en el microcontrolador PIC18F458

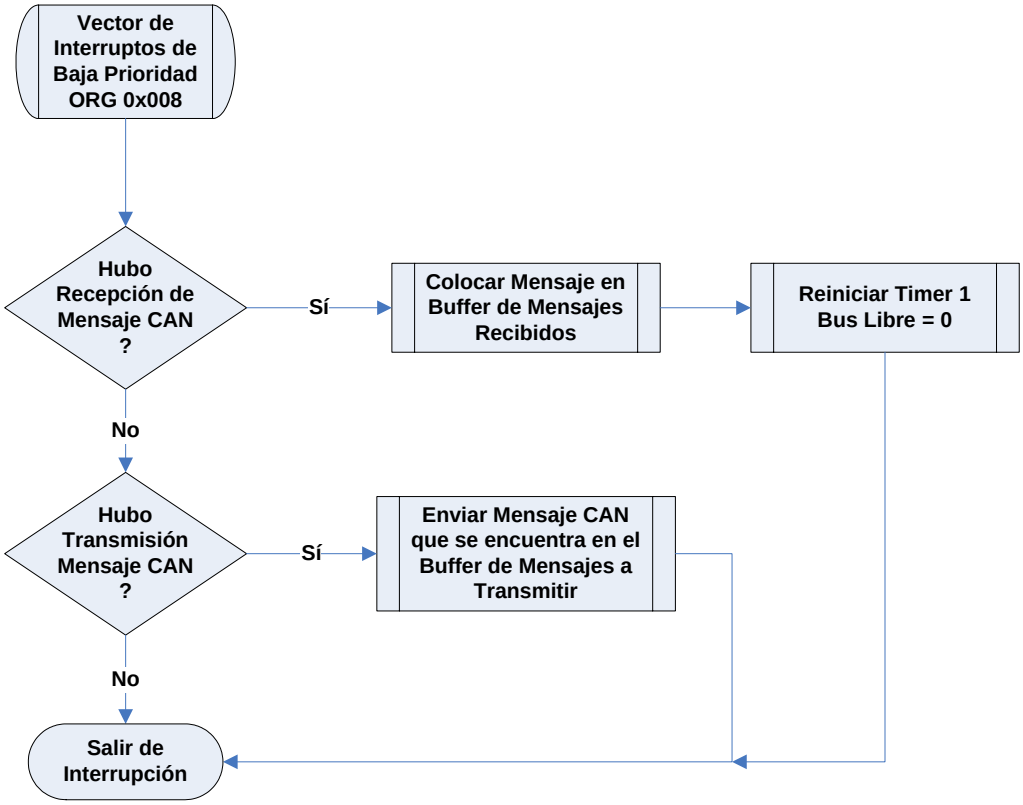


Figura 17: Diagrama de flujo del programa principal en el PIC18F258

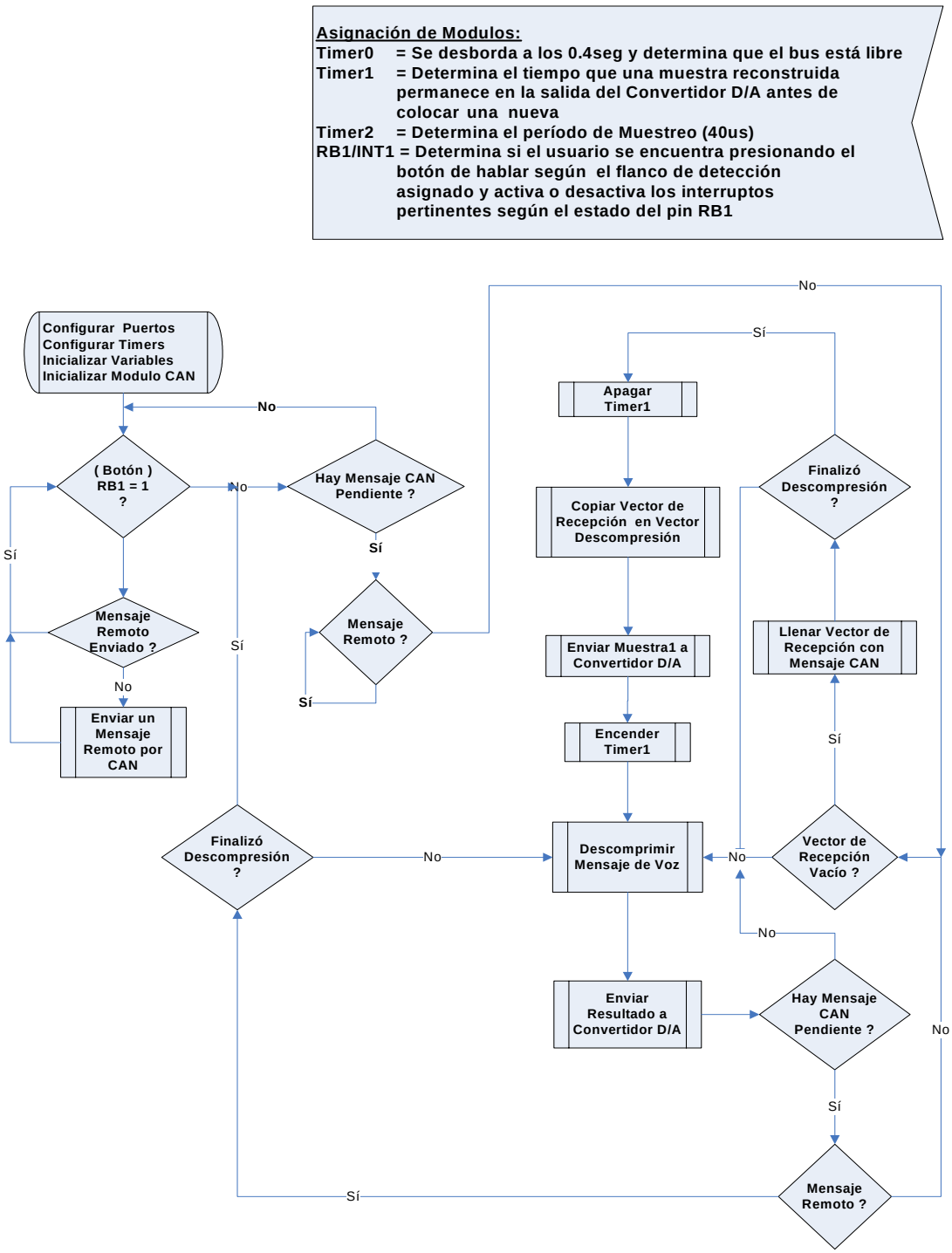


Figura 18: Diagrama de flujo del algoritmo de compresión de datos

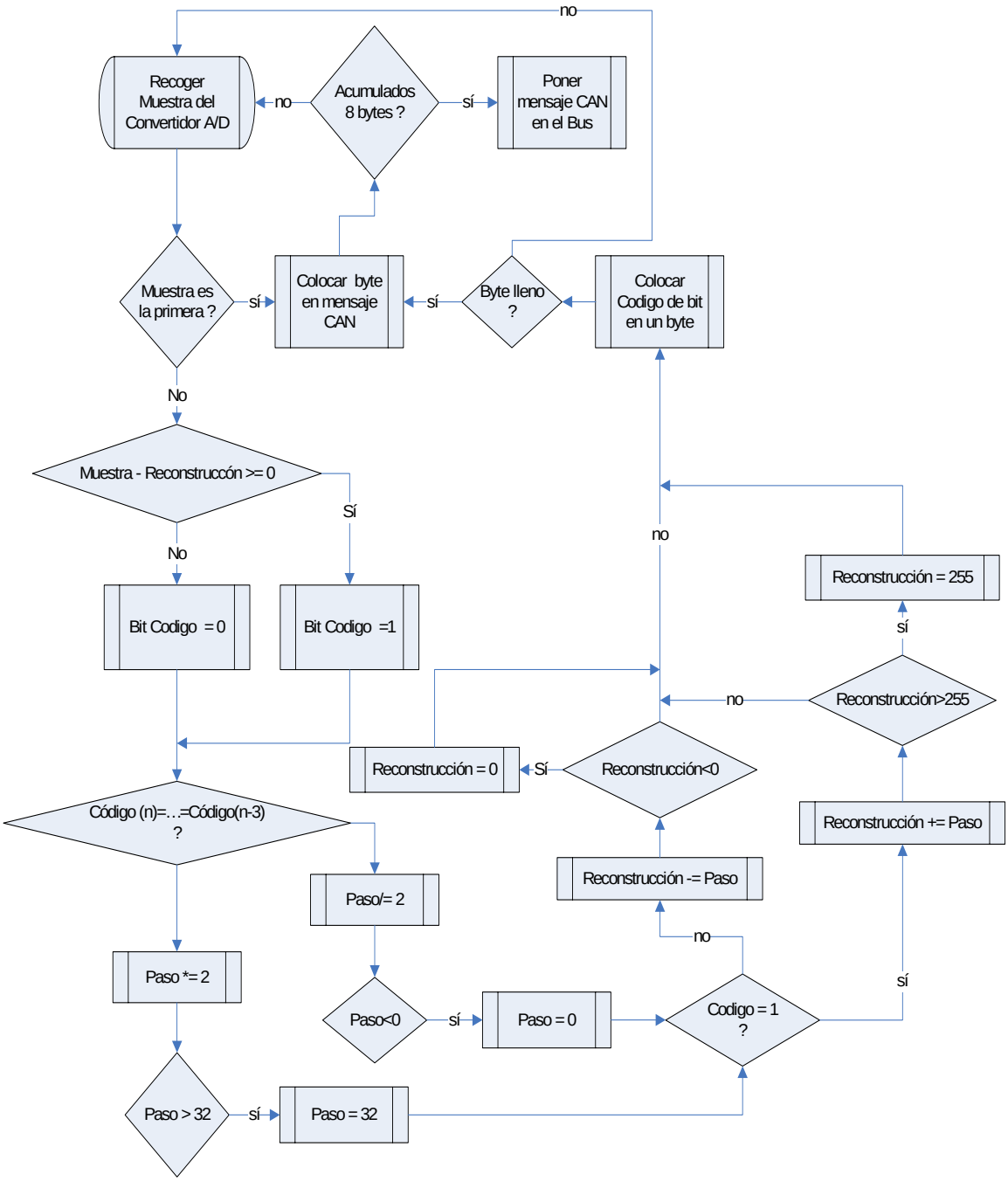
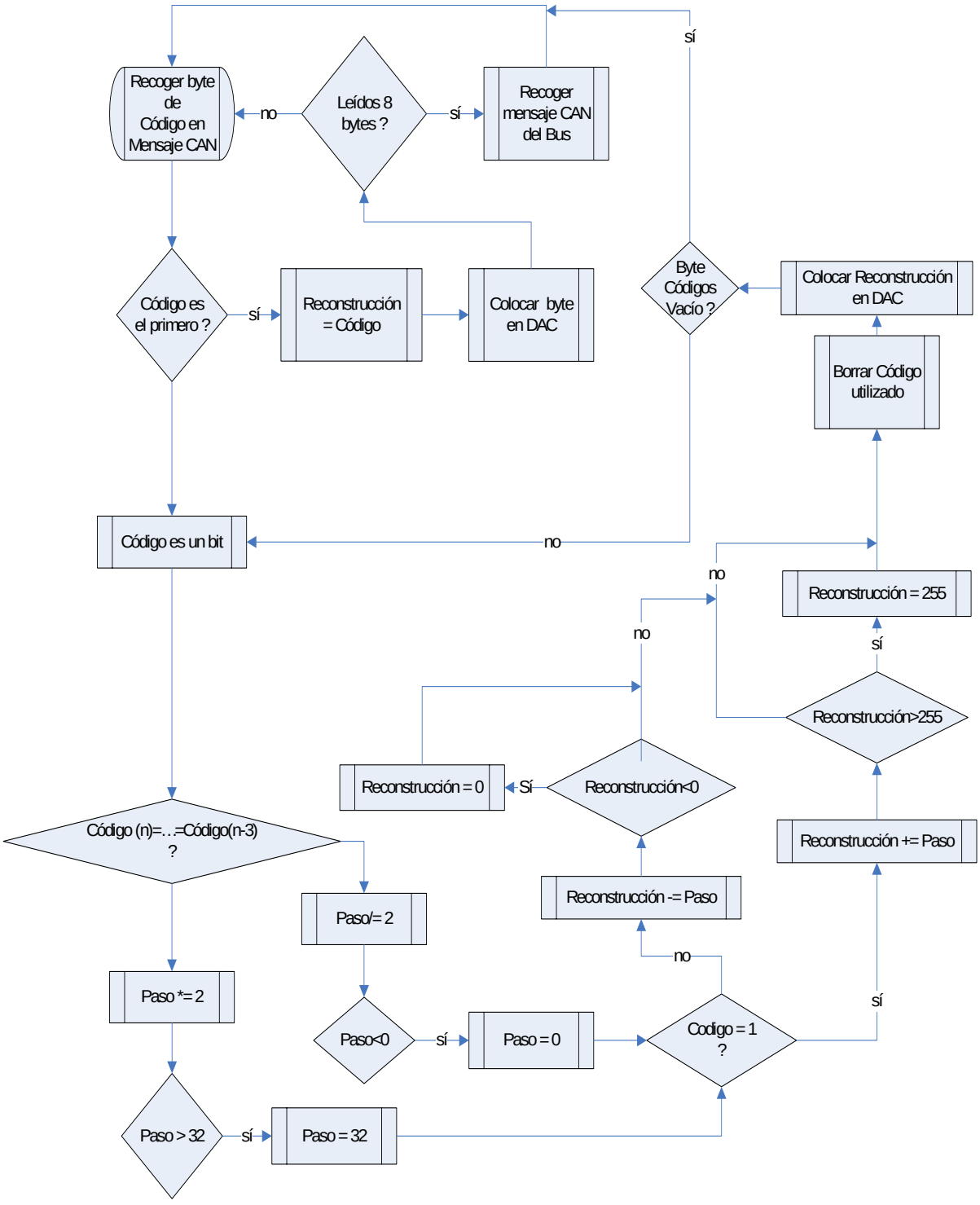


Figura 19: Diagrama de flujo del algoritmo de descompresión de datos



Código Visual BASIC del programa que simula el muestreo y la filtración en la modulación ADPCM

```
Sub Adaptive_CAN() ' Adaptive_CAN Macro ' Macro recorded 1/1/2005 by VELIZ

    Vmax = Cells(1, 14)
    mystep = Cells(1, 2)
    adeltamin = Cells(1, 41)
    adeltamax = Cells(2, 41)
    adelta = adeltamin
    maxsample = Cells(1, 47)
    sample = 0
    For i = mystep To 512 Step mystep
        stprev = Cells(8 + i - mystep, 4)
        skprev = Cells(8 + i - mystep, 50)
        ek = stprev - skprev
        If ek >= 0 Then '(stprev >= sk)
            signo = 1
            bk = 1
        Else
            signo = -1
            bk = 0
        End If
        bk1 = Cells(8 + i - mystep, 49)
        bk2 = Cells(8 + i - 2 * mystep, 49)
        If i >= 12 Then
            bk3 = Cells(8 + i - 3 * mystep, 49)
        End If
        If (bk = bk1) And (bk1 = bk2) And (bk2 = bk3) Then
            If Abs(adelta) < adeltamax Then
                adelta = adelta * 2
            End If
        Else
            If Abs(adelta) > adeltamin Then
                adelta = 1 'adelta / 2
            End If
        End If
        If (sample = 0) Then
            eqk = 0
            adelta = adeltamin
            sk = stprev
            sample = sample + 1
        Else
            eqk = adelta / 255 * Vmax * signo
            sk = skprev + eqk
            sample = sample + 1
            If (sample = maxsample) Then
                sample = 0
            End If
        End If



        Cells(8 + i, 46) = ek
```

```
                Cells(8 + i, 47) = eqk
                Cells(8 + i, 48) = adelta
                Cells(8 + i, 49) = bk
                Cells(8 + i, 50) = sk
            Next i
    Call apply_filtro(51, 52)
End Sub



Sub apply_filtro(ByVal scolumna, ByVal skcolumna)
     For i = 0 To 511
        Integral = 0
          For j = 0 To i
               Integral = integral + Cells(8 + j, scolumna) * Cells(8 + i - j, 31)
          Next j
          Cells(8 + i, skcolumna) = integral
     Next i
End Sub

Sub inv_fftLPF()
'
' fftLPF Macro
' Macro recorded 28/11/2004 by compu2
'
'
    Application.Run "Fourier", ActiveSheet.Range("$AB$8:$AB$519"), _
    ActiveSheet.Range("$AD$8"), True, False
    ActiveWindow.SmallScroll Down:=-15
End Sub
```

Listado de la sección de programa en lenguaje C que define parámetros y maneja el módulo CAN grabado en la memoria de programa del PIC18F458 y PIC18F258

```
#define BAUD_RATE_PRESC 16
#define SJW_TIME 2
#define SAMPLES 3
#define PROP_TIME 3
#define PH_SEG_1 3
#define PH_SEG_2 3
#define CAN_CONFIG_1 BAUD_RATE_PRESC-1|(SJW_TIME-1<<6)
#if SAMPLES == 1
#define CAN_CONFIG_2 0x80|(PH_SEG_1-1<<3)|(PROP_TIME-1)
#elif SAMPLES == 3
#define CAN_CONFIG_2 0xC0|(PH_SEG_1-1<<3)|(PROP_TIME-1)
#else
#error "Number of samples is out of range"
#endif
#define CAN_CONFIG_3 PH_SEG_2-1
#define RXMASK0 2047
#define RXMASK1 2047
#define RXFILT0 10
#define ST_FILTER_0
#define RXFILT1 10
#define ST_FILTER_1
#define RXFILT2 10
#define ST_FILTER_2
#define RXFILT3 10
#define ST_FILTER_3
#define RXFILT4 10
#define ST_FILTER_4
#define RXFILT5 10
#define ST_FILTER_5
//#define LPBACK
#define MY_IDENT 10
#define STD_IDENT
#define RX_BUFFER 8
#define TX_BUFFER 8
//-----------------
#define CAN_LOW_INT
#define CAN_ERROR_HANDLER_ENABLE
```

```
#include "can.h"

#if defined(HI_TECH_C)
   #if defined(_MPC_)
      #define HITECH_C18
   #else
      #error "Unknown part is selected."
   #endif
#else
   #define MCHP_C18
#endif

#if defined(MCHP_C18) && defined(HITECH_C18)
#error "Invalid Compiler selection."
#endif

#if !defined(MCHP_C18) && !defined(HITECH_C18)
#error "Compiler not supported."
#endif


#if defined(HITECH_C18)
#include <pic18.h>

#elif defined(MCHP_C18)
#include <p18cxxx.h>

#define TRISB2 TRISBbits.TRISB2
#define TRISB3 TRISBbits.TRISB3
#define TXB0REQ TXB0CONbits.TXREQ
#define RXB0FUL RXB0CONbits.RXFUL
#define RXB0RTRRO RXB0CONbits.RXRTRRO

#define RXB0IF PIR3bits.RXB0IF
#define RXB1IF PIR3bits.RXB1IF
#define TXB0IF PIR3bits.TXB0IF
#define TXB1IF PIR3bits.TXB1IF
#define TXB2IF PIR3bits.TXB2IF
#define ERRIF PIR3bits.ERRIF
#define WAKIF PIR3bits.WAKIF

#define RXB0IE PIE3bits.RXB0IE
#define RXB1IE PIE3bits.RXB1IE
#define TXB0IE PIE3bits.TXB0IE
#define TXB1IE PIE3bits.TXB1IE
#define TXB2IE PIE3bits.TXB2IE
#define ERRIE PIE3bits.ERRIE
#define WAKIE PIE3bits.WAKIE
#endif
```

```
#define CONFIG_MODE 0x9F
#define LISTEN_MODE 0x7F
#define LOOPBACK_MODE 0x5F
#define DISABLE_MODE 0x3F
#define NORMAL_MODE 0x1F
#define MODE_MASK 0xE0

#define NoInterrupt 0x00
#define ErrorInterrupt 0x02
#define TXB2Interrupt 0x04
#define TXB1Interrupt 0x06
#define TXB0Interrupt 0x08
#define RXB1Interrupt 0x0A
#define RXB0Interrupt 0x0C
#define WakeUpInterrupt 0x0E


#ifdef CAN_ERROR_HANDLER_ENABLE
extern void CANErrorHandler(void);
extern void BUS_busy(void);
#define CAN_INT_BITS 0x3F
#else
#define CAN_INT_BITS 0x1F
#endif

union RXBuffer {
    unsigned char Data[14];
    struct {
        union {
            unsigned char Byte;
            struct {
                unsigned FILHIT0:1;
                unsigned JTOFF:1;
                unsigned RXB0DBEN:1;
                unsigned RXRTRRO:1;
                unsigned :1;
                unsigned RXM0:1;
                unsigned RXM1:1;
                unsigned RXFUL:1;
            } Bits;
        } RXBCON;
        union {
            unsigned char Byte;
        } RXBSIDH;
        union {
            unsigned char Byte;
            struct {
                unsigned EID16:1;
                unsigned EID17:1;
                unsigned :1;
```

```
                unsigned EXID:1;
                unsigned SRR:1;
                unsigned SID0:1;
                unsigned SID1:1;
                unsigned SID2:1;
            } Bits;
        } RXBSIDL;
        union {
            unsigned char Byte;
        } RXBEIDH;
        union {
            unsigned char Byte;
        } RXBEIDL;
        union {
            unsigned char Byte;
            struct {
                unsigned DLC0:1;
                unsigned DLC1:1;
                unsigned DLC2:1;
                unsigned DLC3:1;
                unsigned RB0:1;
                unsigned RB1:1;
                unsigned RXRTR:1;
                unsigned :1;
            } Bits;
        } RXBDLC;
        union {
            unsigned char Array[8];
            struct {
                unsigned char RXBD0;
                unsigned char RXBD1;
                unsigned char RXBD2;
                unsigned char RXBD3;
                unsigned char RXBD4;
                unsigned char RXBD5;
                unsigned char RXBD6;
                unsigned char RXBD7;
            } Bytes;
        } RXBD;
    } Specific;
};

union TXBuffer {
    unsigned char Data[14];
    struct {
        union {
            unsigned char Byte;
            struct {
                unsigned TXPRI0:1;
                unsigned TXPRI1:1;
```

```
            unsigned :1;
            unsigned TXREQ:1;
            unsigned TXERR:1;
            unsigned TXLARB:1;
            unsigned TXABT:1;
        } Bits;
} TXBCON;
union {
    unsigned char Byte;
} TXBSIDH;
union {
    unsigned char Byte;
    struct {
        unsigned EID16:1;
        unsigned EID17:1;
        unsigned :1;
        unsigned EXIDE:1;
        unsigned :1;
        unsigned SID0:1;
        unsigned SID1:1;
        unsigned SID2:1;
    } Bits;
} TXBSIDL;
union {
    unsigned char Byte;
} TXBEIDH;
union {
    unsigned char Byte;
} TXBEIDL;
union {
    unsigned char Byte;
    struct {
        unsigned DLC0:1;
        unsigned DLC1:1;
        unsigned DLC2:1;
        unsigned DLC3:1;
        unsigned :1;
        unsigned :1;
        unsigned TXRTR:1;
        unsigned :1;
    } Bits;
} TXBDLC;
union {
    unsigned char Array[8];
    struct {
        unsigned char TXBD0;
        unsigned char TXBD1;
        unsigned char TXBD2;
        unsigned char TXBD3;
        unsigned char TXBD4;
```

```
                unsigned char TXBD5;
                unsigned char TXBD6;
                unsigned char TXBD7;
            } Bytes;
        } TXBD;
    } Specific;
};
union RXBuffer RXMessage[RXBUF];
union TXBuffer TXMessage[TXBUF];
char RXRPtr = 0;
char RXWPtr = 0;
char TXRPtr = 0;
char TXWPtr = 0;


void CANGetMessage(void);
char CANPutMessage(void);

char CANOpen(unsigned char CONFIG1, unsigned char CONFIG2, unsigned char CONFIG3)
{
    TRISB2 = 0;
    TRISB3 = 1;
    PIE3 = 0;
    PIR3 = 0;
    CANCON = (CONFIG_MODE & MODE_MASK) | (CANCON & 0x3F);
    while((CANSTAT & MODE_MASK) != (CONFIG_MODE & MODE_MASK));

    BRGCON1 = CONFIG1;
    BRGCON2 = CONFIG2;
    BRGCON3 = CONFIG3;

    RXB0CON = 0x04;
    RXB1CON = 0x00;


#ifdef ACCEPT_STANDARD_FILTER_0
    RXF0SIDL = (unsigned char)(ACCEPTANCE_FILTER_0 << 5);
    RXF0SIDH = (unsigned char)(ACCEPTANCE_FILTER_0 >> 3);
#else
    RXF0EIDL = (unsigned char)(ACCEPTANCE_FILTER_0 & 0xFF);
    RXF0EIDH = (unsigned char)((ACCEPTANCE_FILTER_0 >> 8) & 0xFF);
    RXF0SIDL = (unsigned char)((ACCEPTANCE_FILTER_0 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_0 >> 13) & 0xE0) | 0x08;
    RXF0SIDH = (unsigned char)((ACCEPTANCE_FILTER_0 >> 21) & 0xFF);
#endif
#ifdef ACCEPT_STANDARD_FILTER_1
    RXF1SIDL = (unsigned char)(ACCEPTANCE_FILTER_1 << 5);
    RXF1SIDH = (unsigned char)(ACCEPTANCE_FILTER_1 >> 3);
#else
    RXF1EIDL = (unsigned char)(ACCEPTANCE_FILTER_1 & 0xFF);
```

```
    RXF1EIDH = (unsigned char)((ACCEPTANCE_FILTER_1 >> 8) & 0xFF);
    RXF1SIDL = (unsigned char)((ACCEPTANCE_FILTER_1 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_1 >> 13) & 0xE0) | 0x08;
    RXF1SIDH = (unsigned char)((ACCEPTANCE_FILTER_1 >> 21) & 0xFF);
#endif

#ifdef ACCEPT_STANDARD_FILTER_2
    RXF2SIDL = (unsigned char)(ACCEPTANCE_FILTER_2 << 5);
    RXF2SIDH = (unsigned char)(ACCEPTANCE_FILTER_2 >> 3);
#else
    RXF2EIDL = (unsigned char)(ACCEPTANCE_FILTER_2 & 0xFF);
    RXF2EIDH = (unsigned char)((ACCEPTANCE_FILTER_2 >> 8) & 0xFF);
    RXF2SIDL = (unsigned char)((ACCEPTANCE_FILTER_2 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_2 >> 13) & 0xE0) | 0x08;
    RXF2SIDH = (unsigned char)((ACCEPTANCE_FILTER_2 >> 21) & 0xFF);
#endif

#ifdef ACCEPT_STANDARD_FILTER_3
    RXF3SIDL = (unsigned char)(ACCEPTANCE_FILTER_3 << 5);
    RXF3SIDH = (unsigned char)(ACCEPTANCE_FILTER_3 >> 3);
#else
    RXF3EIDL = (unsigned char)(ACCEPTANCE_FILTER_3 & 0xFF);
    RXF3EIDH = (unsigned char)((ACCEPTANCE_FILTER_3 >> 8) & 0xFF);
    RXF3SIDL = (unsigned char)((ACCEPTANCE_FILTER_3 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_3 >> 13) & 0xE0) | 0x08;
    RXF3SIDH = (unsigned char)((ACCEPTANCE_FILTER_3 >> 21) & 0xFF);
#endif

#ifdef ACCEPT_STANDARD_FILTER_4
    RXF4SIDL = (unsigned char)(ACCEPTANCE_FILTER_4 << 5);
    RXF4SIDH = (unsigned char)(ACCEPTANCE_FILTER_4 >> 3);
#else
    RXF4EIDL = (unsigned char)(ACCEPTANCE_FILTER_4 & 0xFF);
    RXF4EIDH = (unsigned char)((ACCEPTANCE_FILTER_4 >> 8) & 0xFF);
    RXF4SIDL = (unsigned char)((ACCEPTANCE_FILTER_4 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_4 >> 13) & 0xE0) | 0x08;
    RXF4SIDH = (unsigned char)((ACCEPTANCE_FILTER_4 >> 21) & 0xFF);
#endif

#ifdef ACCEPT_STANDARD_FILTER_5
    RXF5SIDL = (unsigned char)(ACCEPTANCE_FILTER_5 << 5);
    RXF5SIDH = (unsigned char)(ACCEPTANCE_FILTER_5 >> 3);
#else
    RXF5EIDL = (unsigned char)(ACCEPTANCE_FILTER_5 & 0xFF);
    RXF5EIDH = (unsigned char)((ACCEPTANCE_FILTER_5 >> 8) & 0xFF);
    RXF5SIDL = (unsigned char)((ACCEPTANCE_FILTER_5 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_4 >> 13) & 0xE0) | 0x08;
    RXF5SIDH = (unsigned char)((ACCEPTANCE_FILTER_5 >> 21) & 0xFF);
#endif
```

```
    RXM0EIDL = (unsigned char)(ACCEPTANCE_MASK_0_1 & 0xFF);
    RXM0EIDH = (unsigned char)((ACCEPTANCE_MASK_0_1 >> 8) & 0xFF);
    RXM0SIDL = (unsigned char)((ACCEPTANCE_MASK_0_1 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_MASK_0_1 >> 13) & 0xE0) | 0x08;
    RXM0SIDH = (unsigned char)((ACCEPTANCE_MASK_0_1 >> 21) & 0xFF);
    RXM1EIDL = (unsigned char)(ACCEPTANCE_MASK_2_5 & 0xFF);
    RXM1EIDH = (unsigned char)((ACCEPTANCE_MASK_2_5 >> 8) & 0xFF);
    RXM1SIDL = (unsigned char)((ACCEPTANCE_MASK_2_5 >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_MASK_2_5 >> 13) & 0xE0) | 0x08;
    RXM1SIDH = (unsigned char)((ACCEPTANCE_MASK_2_5 >> 21) & 0xFF);

    CIOCON = 0x20;

#ifdef CAN_LOW_INT
    IPR3 = 0x00;
#else
    IPR3 = 0xFF;
#endif

#ifdef USE_LOOPBACK_MODE
    CANCON = (LOOPBACK_MODE & MODE_MASK) | (CANCON & (MODE_MASK ^ 0xFF));
    while((CANSTAT & MODE_MASK) != (LOOPBACK_MODE & MODE_MASK));
#else
    CANCON = (NORMAL_MODE & MODE_MASK) | (CANCON & (MODE_MASK ^ 0xFF));
    while((CANSTAT & MODE_MASK) != (NORMAL_MODE & MODE_MASK));
#endif
    PIE3 = CAN_INT_BITS & 0xE3;
    PIR3 = 0x18;
//Send my address to notify our presence
#ifdef MY_ADDRESS_IS_STANDARD
    TXB0SIDL = (unsigned char)(MY_ADDRESS_IDENTIFIER << 5);
    TXB0SIDH = (unsigned char)(MY_ADDRESS_IDENTIFIER >> 3);
    TXB0DLC = 0x00;
    TXB0CON = 0x00;
    TXB0REQ = 1;
#else
    TXB0EIDL = (unsigned char)(MY_ADDRESS_IDENTIFIER & 0xFF);
    TXB0EIDH = (unsigned char)((MY_ADDRESS_IDENTIFIER >> 8) & 0xFF);
    TXB0SIDL = (unsigned char)((MY_ADDRESS_IDENTIFIER >> 16) & 0x03) | (unsigned
char)((ACCEPTANCE_FILTER_2 >> 13) & 0xE0) | 0x08;
    TXB0SIDH = (unsigned char)((MY_ADDRESS_IDENTIFIER >> 21) & 0xFF);
    TXB0DLC = 0x00;
    TXB0CON = 0x00;
    TXB0REQ = 1;
#endif

    return 0;
}
```

```
void CANISR(void)
{
    unsigned char TempCANCON;

    if(PIR3 & PIE3)
    {
        TempCANCON = CANCON;

        if(RXB0IF && RXB0IE)
        {
            BUS_busy();
            RXB0IF = 0;
            CANCON = CANCON & 0xF0 | RXB0Interrupt;
            CANGetMessage();
        }
        else if(RXB1IF && RXB1IE)
        {
            BUS_busy();
            RXB1IF = 0;
            CANCON = CANCON & 0xF0 | RXB1Interrupt;
            CANGetMessage();
        }
        else if(TXB0IF && TXB0IE)
        {
            CANCON = CANCON & 0xF0 | TXB0Interrupt;
            if(CANPutMessage())
                TXB0IE = 0;
            else
                TXB0IF = 0;
        }
        else if(TXB1IF && TXB1IE)
        {
            CANCON = CANCON & 0xF0 | TXB1Interrupt;
            if(CANPutMessage())
                TXB1IE = 0;
            else
                TXB1IF = 0;
        }
        else if(TXB2IF && TXB2IE)
        {
            CANCON = CANCON & 0xF0 | TXB2Interrupt;
            if(CANPutMessage())
                TXB2IE = 0;
            else
                TXB2IF = 0;
        }
        else if(ERRIF && ERRIE)
        {
            ERRIF = 0;
#ifdef CAN_ERROR_HANDLER_ENABLE
```

```
        CANErrorHandler();
#endif

        }
        else if(WAKIF && WAKIE)
        {
            WAKIF = 0;
        }

        CANCON = TempCANCON;
    }
}


void CANGetMessage(void)
{
    unsigned char i;
    if(++RXWPtr >= RXBUF)
        RXWPtr = 0;

    for(i=0;i<14;i++)
    {
        RXMessage[RXWPtr].Data[i] = *(unsigned char *)(0xF60+i);
    }
    RXB0FUL = 0;
}

char CANPutMessage(void)
{
    unsigned char i;
    if(TXWPtr != TXRPtr)
    {
        if(++TXRPtr >= TXBUF)
            TXRPtr = 0;
        for(i=0;i<14;i++)
            *(unsigned char *)(0xF60+i) = TXMessage[TXRPtr].Data[i];
        RXB0RTRRO = 1;
        return 0;
    }
    else
        return 1;
}


char CANPut(struct CANMessage Message)
{
    unsigned char TempPtr, i;
    if(TXWPtr == TXRPtr-1 || (TXWPtr == TXBUF-1 && TXRPtr == 0)) //if all transmit buffers are full
return 1
        return 1;
```

```
    if(TXWPtr >= TXBUF-1)
    {
        TempPtr = 0;
    }
    else
    {
        TempPtr = TXWPtr+1;
    }

    if(Message.NoOfBytes > 8)
        Message.NoOfBytes = 8;

    TXMessage[TempPtr].Specific.TXBDLC.Byte = Message.NoOfBytes;

    if(!Message.Remote)
    {
        TXMessage[TempPtr].Specific.TXBDLC.Bits.TXRTR = 0;

        for(i = 0; i < Message.NoOfBytes; i++)
        {
            TXMessage[TempPtr].Specific.TXBD.Array[i] = Message.Data[i];
        }
    }
    else
    {
        TXMessage[TempPtr].Specific.TXBDLC.Bits.TXRTR = 1;
    }
    if(Message.Ext)
    {
        TXMessage[TempPtr].Specific.TXBEIDL.Byte = (unsigned char)(Message.Address & 0xFF);
        TXMessage[TempPtr].Specific.TXBEIDH.Byte = (unsigned char)((Message.Address >> 8) &
0xFF);
        TXMessage[TempPtr].Specific.TXBSIDL.Byte = (unsigned char)((Message.Address >> 16) &
0x03) | (unsigned char)((Message.Address >> 13) & 0xE0);
        TXMessage[TempPtr].Specific.TXBSIDH.Byte = (unsigned char)((Message.Address >> 21) &
0xFF);
        TXMessage[TempPtr].Specific.TXBSIDL.Bits.EXIDE = 1;
    }
    else
    {
        TXMessage[TempPtr].Specific.TXBSIDL.Byte = (unsigned char)((Message.Address << 5) &
0xFF);
        TXMessage[TempPtr].Specific.TXBSIDH.Byte = (unsigned char)((Message.Address >> 3) &
0xFF);
        TXMessage[TempPtr].Specific.TXBSIDL.Bits.EXIDE = 0;
    }
    TXMessage[TempPtr].Specific.TXBCON.Byte = Message.Priority & 0x03;

    TXWPtr = TempPtr;
```

```
    if(!TXB0IE)
        TXB0IE = 1;
    else if(!TXB1IE)
        TXB1IE = 1;
    else if(!TXB2IE)
        TXB2IE = 1;

    return 0;
}
char CANRXMessageIsPending(void)
{
    if(RXWPtr != RXRPtr)
        return 1;
    else
        return 0;
}

struct CANMessage CANGet(void)
{
    struct CANMessage ReturnMessage;
    unsigned char TempPtr, i;


    if(RXRPtr >= RXBUF-1)
    {
        TempPtr = 0;
    }
    else
    {
        TempPtr = RXRPtr+1;
    }

    ReturnMessage.NoOfBytes = RXMessage[TempPtr].Specific.RXBDLC.Byte & 0x0F;
    if(RXMessage[TempPtr].Specific.RXBCON.Bits.RXRTRRO)
    {
        ReturnMessage.Remote = 1;
    }
    else
    {
        ReturnMessage.Remote = 0;
        for(i = 0; i < ReturnMessage.NoOfBytes; i++)
            ReturnMessage.Data[i] = RXMessage[TempPtr].Specific.RXBD.Array[i];
    }
    CANCON = CANCON & 0xF0;

    ReturnMessage.Address = (unsigned int)(RXMessage[TempPtr].Specific.RXBSIDH.Byte) << 3;
    ReturnMessage.Address |= (RXMessage[TempPtr].Specific.RXBSIDL.Byte >> 5);
```

```
if(RXMessage[TempPtr].Specific.RXBSIDL.Bits.EXID)
    {
        ReturnMessage.Ext = 1;
        ReturnMessage.Address = (ReturnMessage.Address << 2) |
(RXMessage[TempPtr].Specific.RXBSIDL.Byte & 0x03);
        ReturnMessage.Address = ReturnMessage.Address << 16;
        ReturnMessage.Address |= (unsigned int)(RXMessage[TempPtr].Specific.RXBEIDH.Byte) << 8;
        ReturnMessage.Address |= RXMessage[TempPtr].Specific.RXBEIDL.Byte;
    }
    else
    {
        ReturnMessage.Ext = 0;
    }

    RXRPtr = TempPtr;
    return ReturnMessage;
}


void CANSetMode(unsigned char Mode)
{
#ifdef USE_LOOPBACK_MODE
    if(Mode == NORMAL_MODE)
    {
        CANCON = (LOOPBACK_MODE & MODE_MASK) | (CANCON & (MODE_MASK ^ 0xFF));
        while((CANSTAT & MODE_MASK) != (LOOPBACK_MODE & MODE_MASK));
    }
    else
#endif
    {
        CANCON = (Mode & MODE_MASK) | (CANCON & (MODE_MASK ^ 0xFF));
        while((CANSTAT & MODE_MASK) != (Mode & MODE_MASK));
    }

}
//-----------------------------------------------------------------------
unsigned char RXNumber(void)
{
    return RXWPtr;
}
```

Listado de la sección de programa en lenguaje C grabado en la memoria de programa del PIC18F458

```
#include  <p18f458.h>    //incluir librería de nombres de registros del PIC
#include "can.h"        //incluir librería de manejo bus CAN
#include <delays.h>        //incluir librería de demoras
#include <adc.h>       //incluir librería de manejo Convertidor A/D

//*****************************************************************************
//VARIABLES GLOBALES
const unsigned char  sampleperiod = 98, spicommand = 0b00110000;
const unsigned short timer3delay = 25000, timer0delay = 0,timer1delay = 65340;//40 µs = 65340
unsigned short ADCresult;
unsigned char ADPCMbitcount = 0,CANbytefull = 0;
struct CANMessage RX_Message, TX_Message;
unsigned char i,j;
volatile unsigned char CANbytecount = 0, adaptive_step = 1, adaptive_sign = 0, SUMADC;
volatile unsigned char infobyte, ADPCMbyte[8],ReceiveVector[8],mensajes=0;
volatile unsigned char CAN2bytecount = 0, adaptive2_step = 1, adaptive2_sign = 0, SUM2ADC;
volatile unsigned char info2byte, ADPCM2byte[8],messageout;
volatile unsigned char bitcount, pastbits, received;
volatile unsigned char direccion, olddireccion = 0, DACcontrol=0, DAC=0;
volatile unsigned char spidata1=0b00110000, spitemp;
volatile unsigned char origin, busfree = 0 ,CANdataempty = 1;
void ADC_Int_Routine(void);   //Interrupción del ADC que recoge muestras y al s coloca en un byte CAN
void TMR0_Int_Routine(void); //Define si  el bus CAN está libre luego de 0.4 seg.
void TMR1_Int_Routine(void); //Define el tiempo que una muestra permanece en el DAC
void TMR2_Int_Routine(void); //Acciona el ADC y determina la frecuencia de muestreo (25kHz)
void TMR3_Int_Routine(void); //Define el tiempo que se escucha el timbre
void RB1_Int_Routine(void);    //Establece si el modulo está en modo escuchar o hablar
void Send_CAN_Message(void);    //Coloca un mensaje CAN en la cola de envío
void GET_ADPCM_STEP(void);   //Rutina de Compresión de Datos
void PUT_ADC_IN_DAC(void);    //Determina paso a utilizar y si sumarlo o restarlo (Compresor)
void DET_ADPCM_STEP(void);   //Determina paso a utilizar y si sumarlo o restarlo (Descompresor)
void REFRESH_DAC(void);     //Rutina de descompresión de Datos
void DIPSWITCH(void);          //Recoge dirección CAN del Puerto D
void PRODUCE_RING(void);  //Coloca la dirección del Timbrte a activar en el Puerto B
void SPI_DAC(unsigned char); //Envía serialmente al DAC el dato descomprimido
void Get_Request(void);      //Recoge Mensajes CAN del Buffer de entrada
void Send_CAN_Request(void);     //Envía Mensaje Remoto (sin datos)
void SET_CAN_HEADER(void);   //Establece que los mensajes CAN tineen 8 datos y que son standard
void REMOTE_MESSAGE(void); //Activa el timbre si el mensaje CAN recogido es remoto
//*****************************************************************************
```

```
//*****************************************************************************
//LOW PRIORITY INTERRUPT ROUTINES
#pragma   interrupt InterruptHandlerLow
void InterruptHandlerLow(void)
{
  CANISR();                    //Rutina de Interrupciones de baja prioridad
}

//-----------------------------------------------------------------------
// HIGH PRIORITY INTERUPT ROUTINES
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh (void)
{
//Interrupt ADC
   if (PIR1bits.ADIF&&PIE1bits.ADIE)            //Interrupt ADC
   {
        ADC_Int_Routine();
        PIR1bits.ADIF = 0;            //Borrar Bandera ADC
   }
//Interrupt Timer2
    else if (PIR1bits.TMR2IF&&PIE1bits.TMR2IE)   //Interrupt Timer2
    {
        TMR2_Int_Routine();
        PIR1bits.TMR2IF = 0;        //Borrar Bandera TMR2
    }
//Interrupt Timer1
    else if (PIR1bits.TMR1IF&&PIE1bits.TMR1IE) //Interrupt Timer1
    {
        TMR1_Int_Routine();
        PIR1bits.TMR1IF = 0;        //Borrar Bandera TMR1
    }
//Interrupt Timer3
    else if (PIR2bits.TMR3IF&&PIE2bits.TMR3IE) //Interrupt Timer3
    {
        TMR3_Int_Routine();
        PIR2bits.TMR3IF = 0;         //Borrar Bandera TMR3
    }
//Interrupt Timer0
    else if (INTCONbits.TMR0IF)        //Interrupt Timer0
    {
        TMR0_Int_Routine();
        INTCONbits.TMR0IF = 0;   //Borrar Bandera TMR0
    }
//Interrupt RB1
    else if  (INTCON3bits.INT1IF)       //Interrupt RB1
   {
        RB1_Int_Routine();
        INTCON3bits.INT1IF = 0;   //Borrar Bandera INT1
   }
}//end InterruptHandlerHigh
```

```
//---------------------------------------------------------------------------
// Vector de Interrupciones de Baja Prioridad
#pragma code InterruptVectorLow = 0x018
void InterruptVectorLow (void)
{
    _asm
        goto InterruptHandlerLow
    _endasm
}
//---------------------------------------------------------------------------
// Vector de Interruptos de Alta Prioridad
#pragma code InterruptVectorHigh = 0x008
void InterruptVectorHigh (void)
{
    _asm
        goto InterruptHandlerHigh
    _endasm
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
                            //CODIGO
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
#pragma code
//---------------------------------------------------------------------------
void TMR3_Int_Routine(void)
{
    T3CONbits.TMR3ON   = 0;      //Apagar TIMER3
    LATB        = PORTB & 0b00001111;  //Borrar direccion timbre
    TMR3      = timer3delay;          //Recargar TIMER3
}
//---------------------------------------------------------------------------
void TMR2_Int_Routine(void)
{
    T2CONbits.TMR2ON   = 0;      //Apagar TIMER2
    ConvertADC();                //Iniciar Conversión AD                    Get_Request();
    //Recoger Mensaje Remoto
    TMR2                  = 0;       //Recargar TIMER2
}


//---------------------------------------------------------------------------
void TMR0_Int_Routine(void)
{
    T0CONbits.TMR0ON = 0;        //Apagar TIMER1
    busfree              = 1;         //Asumir Bus Libre
    LATEbits.LATE1       = 1;         //Encender Luz Verde Bus Libre
    LATEbits.LATE2       = 0;         //Apagar Luz Roja Bus Ocupado
    TMR0                 = 0;         //Reiniciar Timer0
    LATCbits.LATC7       = 0;         //Desactivar Filtro de Salida
}
//---------------------------------------------------------------------------
```

```
void TMR1_Int_Routine(void)
{
    T1CONbits.TMR1ON   = 0;  //Apagar TIMER1
    TMR1 = timer1delay;          //Recargar TIMER1
    REFRESH_DAC();        //Descomprimir Dato CAN
    SPI_DAC(DAC);           //Poner Dato Nuevo en DAC
    if (CANdataempty == 0)          //Si no hay mensaje que descomprimir
    T1CONbits.TMR1ON   = 1;      //Encender Timer1 nuevamente
}
//-------------------------------------------------------------------------
void SPI_DAC(unsigned char spidata2)
{
    spitemp    = spidata2>>4;              //MSBits de DATO van en LSBits de 1er byte SPI
    spitemp       = spitemp&0b00001111;       //Borrar MSBits
    spidata1  += spitemp;              //1er byte lleva MSB del dato y Palabra de Config.
    LATAbits.LATA5    = 0;          //CS = 0    //Chip Select DAC = 0
    SSPBUF     = spidata1;       //0010-1111//Cargar Buffer SPI con 1er. byte SPI
    spidata2   = spidata2<<4;              //Poner dato DAC en MSBits de 2do. byte SPI
    Delay1TCY();                  //Esperar  un Ciclo de Reloj = NOP
    spidata1   = spicommand;              //Recargar Palabra Configuración en 1er. byte
    SSPBUF     = spidata2;       //10101010//Cargar Buffer SPI con 2do. byte SPI
    Delay1TCY();                  //NOP 1
    Delay1TCY();                  //NOP 2
    Delay1TCY();                  //NOP 3
    Delay1TCY();                  //NOP 4
    Delay1TCY();                  //NOP 5
    Delay1TCY();                  //NOP 6
    Delay1TCY();                  //NOP 7
    Delay1TCY();                  //NOP 8
    LATAbits.LATA5    = 1;              //Chip Select DAC = 1
}
//-------------------------------------------------------------------------
void RB1_Int_Routine(void)
{
    if ((INTCON2bits.INTEDG1 == 0)&&(busfree==1))  //Interrupt cayó en flanco de bajada
    {
        INTCON2bits.INTEDG1 = 1;   //Detectar Flanco Subida
        T1CONbits.TMR1ON    = 0;        //Apagar TIMER1
        PIE1bits.TMR1IE          = 0;   //Deshabilitar Interrupt TIMER1
        PIR1bits.TMR1IF          = 0;   //Borrar Bandera TIMER1
        CANbytecount             = 0;   //Borrar Contador de byte a comprimir
        PIE1bits.ADIE            = 1;   //Habilitar Interrupt ADC
        PIE1bits.TMR2IE          = 1;   //Habilitar Interrupt TIMER2
        TMR2_Int_Routine();         //Encender TIMER2
        SET_CAN_HEADER();      //Establecer Parametros de Envío CAN
        PORTEbits.RE0       = 1;    //Encender Luz Amarilla de Hablando
        LATCbits.LATC6    = 1;    //Activar Filtro de Entrada
    }
```

```
    else if(INTCON2bits.INTEDG1 == 1)  //Interrupt cayó en flanco de subida
    {
        INTCON2bits.INTEDG1 = 0;    //Detectar Flanco de Bajada
        T2CONbits.TMR2ON    = 0;       //Apagar TIMER2
        PIE1bits.TMR2IE     = 0;    //Deshabilitar Interrupt TIMER2
        PIR1bits.TMR2IF     = 0;    //Borrar Bandera TIMER2
        ADCON0bits.GO_DONE  = 0;  //Detener Conversión AD
        PIE1bits.ADIE       = 0;    //Deshabilitar Interrupt ADC
        PIR1bits.ADIF       = 0;        //Borrar Bandera ADC
        PIE1bits.TMR1IE     = 1;    //Habilitar Interrupt TIMER1
        CANdataempty        = 1;    //Definir que no hay datos para descomprimir
        CAN2bytecount       = 0;    //Borrar Contador de byte a descomprimir
        PORTEbits.RE0       = 0;    //Apagar Luz Amarilla de Hablando
        LATCbits.LATC6      = 0;    //Desactivar Filtro de Entrada
    }
}
//---------------------------------------------------------------------------
void ADC_Int_Routine(void)
{
    T2CONbits.TMR2ON   = 1;      //Encender TIMER2
    ADCresult = ReadADC();              //Leer Resultado AD (16bits)
    ADCresult>>=8;              //Correr Resultado 8 posiciones hacia LSb
    if (CANbytecount > 0)
    {
        infobyte = infobyte>>1;//Correr 1 pos en byte de Resultados Comprimidos hacia LSb
        if (ADCresult > DACcontrol)
        {
            infobyte+=128;    //Si Resultado AD>Resultado Adaptativo bit comprimido = 1
            adaptive_sign = 1;//El cambio se dio en dirección de aumento
        }//endif
        else adaptive_sign = 0;   //Si Resultado Adaptativo<Resultado AD bitl comprimido = 0
            ADPCMbitcount++;//Incrementar puntero de byte de bits/datos comprimidos
        if (ADPCMbitcount == 8)
        {
            CANbytefull   = 1;//Byte Comprimido está lleno si ya se juntaron 8 bits
            ADPCMbitcount = 0;          //Borrar ountero de byte de bits/datos             }
    }
    else
    {
        infobyte = ADCresult;         //Si es el primer resultado A/D ponerlo en mensaje
    CANbytefull = 1;           //Definir que el byte actual está lleno
    }
//CARGAR BYTE EN CANBUFFER
    if (CANbytefull == 1)              //Si el byte actual está lleno
    {
        ADPCMbyte[CANbytecount]= infobyte;  //Colocar byte lleno en mensaje CAN
        CANbytecount++;         //Incrementar contador de bytes en mensaje
```

```
        if (CANbytecount > 7)         //Si el contador de mensajes es amyor a 7
        {
            Send_CAN_Message();  //Enviar mensaje con 8 bytes
            CANbytecount   = 0; //Reiniciar contador de mensajes
        }
        CANbytefull   = 0;    //Definir que byte actual de datos está vacío
        ADPCMbitcount = 0;      //Reiniciar contador de bits en byte comprimido
    }
    PUT_ADC_IN_DAC();       //Simular que se pone un dato en el DAC

//---------------------------------------------------------------------------
void Send_CAN_Message(void)
{
    for (i=0;i<8;i++)                 //Cargar Mensaje CAN de 8 bytes en otro vector
    {
        TX_Message.Data[i] = ADPCMbyte[i];
    }
    messageout = CANPut(TX_Message);//Poner mensaje en  FIFO de Envío CAN
}
//---------------------------------------------------------------------------
void GET_ADPCM_STEP(void)
{
    SUMADC = (infobyte&128) + (infobyte&64)+ (infobyte&32) + (infobyte&16); //Sumar bits anteriores
    if ((SUMADC==240)||(SUMADC==0))     adaptive_step *=  2;  //Duplicar paso anterior
    else                        adaptive_step  =  1;   //Regresar a paso =  1
    if    (adaptive_step>=64)      adaptive_step  = 32;  //Usar paso máximo
    else if (adaptive_step<1)      adaptive_step  =  1;   //Usar paso mínimo
}
//---------------------------------------------------------------------------
void PUT_ADC_IN_DAC(void)
{
    if ((CANbytecount ==1)&&(ADPCMbitcount==0))//Si es el primer byte de los 8 a enviar
    {
        DACcontrol = infobyte;       //poner byte completo de conversion en DAC
        adaptive_step =  1;         //Regresar paso a unitario
        infobyte      = 42;          //Forzar serie de valores anteriores a 10101010
    }
    else
    {
```

```
        GET_ADPCM_STEP();       //Definir paso a utilizar
        if (adaptive_sign == 1)         //Si es un aumento sumar paso delta
        {
            if ( DACcontrol <(0XFF-adaptive_step)) DACcontrol += adaptive_step;
            else DACcontrol = 0XFF;    //Saturar hacia arriba
        }
        else                    //Si es un disminución restar paso delta
        {
            if (DACcontrol > adaptive_step) DACcontrol -= adaptive_step;
            else DACcontrol = 0X00;          //Saturar hacia abajo
        }
    }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void DET_ADPCM_STEP(void)
{
    SUM2ADC = (pastbits&128) + (pastbits&64) + (pastbits&32) + (pastbits&16);
    if ((SUM2ADC == 240 )||(SUM2ADC == 0)) adaptive2_step<<=  1; //Duplicar paso
    else                            adaptive2_step  =  1; //Volver Paso Unitario
    if     (adaptive2_step >= 64)        adaptive2_step  = 32; //Usar paso máximo
    else if (adaptive2_step <   1)   adaptive2_step  =  1; //Usar paso mínimo
}
//-------------------------------------------------------------------------------
void REFRESH_DAC(void)
{
    if (CAN2bytecount == 0)                 //Si es el primer byte del mensaje
    {
        DAC = ADPCM2byte[CAN2bytecount];//Estado DAC es actualizado con primer byte
    pastbits = 42;      //Trama anterior forzada a ser 10101010
        adaptive2_step = 1;   //Reducir paso delta a la unidad
        CAN2bytecount++;   //Aumentar contador de bytes CAN
        info2byte =    ADPCM2byte[CAN2bytecount];//Recoger siguiente byte  del buffer
        bitcount = 1;                       //Reiniciar contador de muestras de byte
    }

    else
    {
        pastbits>>=1;           //Mover bits de muestra anteriores una posición
        adaptive2_sign = info2byte & bitcount;//Obtener el bit de muestra actual
        if (adaptive2_sign > 0)
        {
            adaptive2_sign = 1;   //Bit de muestra actual es uno
            pastbits +=128;         //Actualizar byte muestras anteriores con bit actual
        }
        else adaptive2_sign = 0;       //Bit de muestra actual es cero
        DET_ADPCM_STEP();       //Determinar paso delta a utilzar
```

```
        if (adaptive2_sign == 1)
        {
            if ( DAC <(0XFF - adaptive2_step)) DAC += adaptive2_step; //Sumar paso
            else DAC = 0XFF;  //Saturar hacia arriba
        }
        else
        {
            if (DAC > adaptive2_step) DAC -= adaptive2_step;    //Restar paso delta
            else DAC = 0X00;                    //Saturar hacia abajo
        }
        bitcount<<=1;           //Duplicar Contador de muestras en byte
        if (bitcount == 0)      //Si contador de bits llegó al final, reiniciarlo
        {
            bitcount = 1;   //Reiniciar Contador de muestras en byte
            CAN2bytecount++;//Aumentar contador de byte CAN a leer
            if (CAN2bytecount > 7)  //Si  contador mayor que 7 reiniciar variables
            {
                CAN2bytecount = 0;//Reiniciar contador de byte CAN a leer
                CANdataempty  = 1;//Definir que ya no hay mas datos a descomprimir                }
            else  info2byte = ADPCM2byte[CAN2bytecount]; //Actualizar byte de muestras
        }
    }
}//endroutine
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void ADC_CONFIG( void )
{
// configure A/D convertor
    OpenADC( ADC_FOSC_64 & ADC_LEFT_JUST & ADC_1ANA_2REF, ADC_CH0 &
ADC_INT_ON );
    ADCON0bits.GO_DONE = 0;        //Detener Conversión AD
    PIE1bits.ADIE       = 0;  //Deshabilitar Interrupción por ADC
    PIR1bits.ADIF       = 0;    //Borrar Bandera Interrupción ADC
    IPR1bits.ADIP       = 1;       //Interrupción AD es de alta prioridad
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void RB1_CONFIG(void)
{
//RB1_INT
    TRISBbits.TRISB1    = 1;    //Puerto RB1 es un entrada digital
    INTCON2bits.RBPU = 0;    //Encender Pull-Ups de Puerto B
    INTCON2bits.INTEDG1  = 0;   //Interrupción INT1 activada con flanco de caída
    INTCON3bits.INT1IP   = 1;//Interrupción INT1 es de alta prioridad
    INTCON3bits.INT1IF   = 0;//Borrar bandera de Interrupción INT1
    INTCON3bits.INT1IE   = 1;//Habilitar Interrupción externa INT1
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
void TIMER0_CONFIG (void)
{
//OpenTimer0( TIMER_INT_ON & T0_8BIT & T0_SOURCE_INT & T0_PS_1_256 );
//config T0CON
    T0CONbits.TMR0ON   = 0; //Apagar Timer 0
    T0CONbits.T08BIT   = 0;         //Timer 0 es un 16 bit timer/counter
    T0CONbits.T0CS     = 0;         //Fuente de Reloj es interna Fosc/4
    T0CONbits.PSA      = 0;      //Asignar Prescaler
    T0CONbits.T0PS0    = 1;         //Prescaler Select 0.1µ*64*65535 = 0.4194 s.
    T0CONbits.T0PS1    = 0;         //Prescale Value = 1:64 (0.4 seg.)
    T0CONbits.T0PS2    = 1;
//enable TMR0 Overflow Interrupt
    INTCON2bits.TMR0IP = 1; //Timer0 tieneInterrupción de Alta Prioridad
    INTCONbits.TMR0IF  = 0;  //Borrar bandera de Interrupción Timer 0
    INTCONbits.TMR0IE  = 1;  //Habilitar Interrupción de Timer 0
//reset TMR0
    TMR0 = timer0delay;          //Cargar Valor Inicial de Timer 0
    T0CONbits.TMR0ON   = 1; //Encender Timer 0;
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TIMER1_CONFIG (void)
{
//config T1CON
    T1CONbits.RD16     = 1;         //Habilitar 16 bit read/write
    T1CONbits.T1CKPS0  = 0;  //Prescale Value = 1:1
    T1CONbits.T1CKPS1  = 0;
    T1CONbits.T1OSCEN  = 0;//Opción de Oscilador externo deshabilitada
    T1CONbits.T1SYNC   = 1;
    T1CONbits.TMR1CS   = 0;  //Fuente de Reloj es interna Fosc/4
//enable TMR1 Overflow Interrupt
    IPR1bits.TMR1IP    = 1;      //Interrupción de Timer1 es de Alta Prioridad
    PIR1bits.TMR1IF    = 0;      //Borrar Bandera de Interrupción Timer1
    PIE1bits.TMR1IE    = 1;      //Interrupción de Timer1  habilitada
//reset TMR1                //65535-6340 = 195
    TMR1 = timer1delay;          //195*0.1µs = 19.5µs
    T1CONbits.TMR1ON   = 0;          //Apagar Timer1
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TIMER2_CONFIG (void)
{
//POSTSCALER
    T2CONbits.TOUTPS0  = 1;      //Prescaler = 1:2
    T2CONbits.TOUTPS1  = 0;      //Timer2 se desborda cada 98*2*0.1µs =19.6µs
    T2CONbits.TOUTPS2  = 0;
    T2CONbits.TOUTPS3  = 0;
//PRESSCALER
    T2CONbits.T2CKPS0  = 0;
    T2CONbits.T2CKPS1  = 0;      //Postscaler = 1:1
```

```
//PERIOD
    PR2 = sampleperiod;          //Sampleperiod  = 98
//INTERRUPT
    PIR1bits.TMR2IF       = 0;       //Borrar Bandera de Interrupción Timer 2
    PIE1bits.TMR2IE       = 0;       //Deshabilitar Interrupción por Timer 2
    IPR1bits.TMR2IP    = 1;          //Timer2 is High Priority
//TMR2 IS OFF
    TMR2 = 0;                  //Inicializar Timer 2
    T2CONbits.TMR2ON  = 0;       //Apagar Timer 2
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TIMER3_CONFIG (void)
{
//TMR3 = OFF
    T3CONbits.TMR3ON  = 0; //Aapgar Timer 3
//16 bit counter
    T3CONbits.RD16    = 1;      //Permitir Escritura/Lectura de 16 bit
//Prescaler
    T3CONbits.T3CKPS0 = 1;  //Prescale Value = 1:8
    T3CONbits.T3CKPS1 = 1;  // Desborde de Timer 3 ocurre = 0.1µs*40k*8 = 32ms
//Count Source
    T3CONbits.TMR3CS  = 0;  //Fuente de conteo es interna: Fosc/4
//Initialize Timer3
    TMR3       =  timer3delay;       //Inicializar Timer3
//INTERRUPT TMR3
    PIR2bits.TMR3IF   = 0;       //Borrar Bandera de Interrupción Timer3
    IPR2bits.TMR3IP   = 1;       //Timer3 es Interrupción de Alta Prioridad
    PIE2bits.TMR3IE   = 1;       //Habilitar Interrupción de Timer3
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void DEFINE_PORTS(void)
{
    CMCON = 0X07;               //Apagar Comparadores RD0:RD3 = 0x00
    TRISD = 255;               //Puerto D es sólo de entrada digitales
//Canal Analógico y Referencias de Voltaje
    TRISAbits.TRISA0 = 1;       //Habilitar Canal 0
    TRISAbits.TRISA2 = 1;       //RA2 es Vref -
    TRISAbits.TRISA3 = 1;       //RA3 es Vref +
//RE0 = Hablando, RE1 = Bus libre, RE2 = Bus Ocupado
    TRISE = 0;              //TRISE  = Outputs
    PORTEbits.RE0 = 0;      //RE0    = LED Hablando
    PORTEbits.RE1 = 0;      //RE1    = LED Bus Libre
    PORTEbits.RE2 = 1;      //RE2    = LED Bus Ocupado
//Salidas Timbre
    TRISAbits.TRISA1 = 0;       //Salida Timbre
    TRISAbits.TRISA4 = 0;       //Salida Timbre
```

```
//Salidas Timbre
    TRISBbits.TRISB4 = 0;      //Dirección Timbre D0
    TRISBbits.TRISB5 = 0;      //Dirección Timbre D1
    TRISBbits.TRISB6 = 0;      //Dirección Timbre D2
    TRISBbits.TRISB7 = 0;      //Dirección Timbre D3
//Estado Timbre
    LATAbits.LATA1 = 0;
    LATAbits.LATA4 = 0;
//Estado Timbre
    LATB       = 0;      //Dirección Timbre D0:D3 = 0
//Filtros
    TRISCbits.TRISC6 = 0;      //Salida Activadora de Filtro de Entrada
    TRISCbits.TRISC7 = 0;      //Salida Activadora de Filtro de Salida
    LATCbits.LATC6  = 0;       //Filtro de Entrada  Inactivo
    LATCbits.LATC7  = 0;       //Filtro de Salida   Inactivo
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void SPI_CONFIG(void)
{
//SPI PORTS
    TRISAbits.TRISA5  = 0;         //CHIP SELECT del SPI (SS)
    TRISCbits.TRISC3  = 0;         //SPI_CLK - Reloj del SPI
    TRISCbits.TRISC5  = 0;         //SPI_SDOUT - Salida Serial SPI
//SPI Iterrupt
    PIE1bits.SSPIE     = 0;        //Deshabilitar Interrupción SPI
    PIR1bits.SSPIF     = 0;        //Borrar Bandera de Interrrupción SPI
    IPR1bits.SSPIP    = 0;         //Establecer Interrupción SPI como de Alta prioridad
//STATUS REGISTER
    SSPSTATbits.SMP  = 0;      //Muestrear Datos a la mitad del bit
    SSPSTATbits.CKE  = 0;      //Transmitir Datos en el flanco de subida
//CONTROL REGISTER
    SSPCON1bits.WCOL  = 0;  //Borrar Bit de Estado Colisiones
    SSPCON1bits.SSPOV = 0;  //Borrar Bit de Desborde
//SPI MODE: Fosc/4 = 10 MHz
    SSPCON1bits.SSPM0 = 0;   //Modo de Operación del SPI es maestro a 10bits/s.
    SSPCON1bits.SSPM1 = 0;
    SSPCON1bits.SSPM2 = 0;
    SSPCON1bits.SSPM3 = 0;
//CONFIGURE SDI, SDO, CLK, CS
    SSPCON1bits.CKP   = 1;         //Estado pasivo de línea de reloj es un uno lógico
    SSPCON1bits.SSPEN = 1;   //Encender módulo SPI
}
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void SET_CAN_HEADER(void)
{
    TX_Message.Address = MY_ADDRESS_IDENTIFIER;//Establecer el identidficador del mensaje
    #ifdef MY_ADDRESS_IS_STANDARD
    TX_Message.Ext = 0;     //Si el identificador es standard, Ext. = 0
    #else
    TX_Message.Ext = 1;     //Si el identificador es extendido, Ext. = 1
    #endif
    TX_Message.NoOfBytes = 8;//Definir número de bytes a enviar
    TX_Message.Remote    = 0;  //Borrar Bandera de Mensaje Remoto
    TX_Message.Priority  = 0;   //Prioridades FIFO de CAN  0->mínima prioridad, 3->máxima
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Get_CAN_Message(void)
{
    if(CANRXMessageIsPending())       //Chequear si hay mensajes sin leer
    {
        RX_Message = CANGet();    //Recuperar Mensaje CAN desde el Buffer
        if ((RX_Message.Remote == 0))//&&(RX_Message.Address > 20))
        {
                for(j=0; j<RX_Message.NoOfBytes; j++) //Cargar Mensaje en Vector de Recepción
                {
                    ReceiveVector[j] = RX_Message.Data[j] ;
                }
                received = 1;
        }
        else REMOTE_MESSAGE();    //Chequear si mensaje es remoto
    }
}//endroutine
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Get_Request(void)
{
    if(CANRXMessageIsPending()) // Si hay mensaje CAN pendiente de Lectura
    {
        RX_Message = CANGet();  //Recuperar Mensaje CAN pendiente
        REMOTE_MESSAGE();          //Chequear si mensaje es remoto
    }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Send_CAN_Request(void)
{
    TX_Message.Ext           = 0; //Si el identificador es standard, Ext. = 0
    TX_Message.NoOfBytes= 0; //Definir número de bytes a enviar
    TX_Message.Remote      = 1; //Activar Bandera de Mensaje Remoto
    TX_Message.Priority = 3; //Prioridades FIFO de CAN  0->mínima, 3->máxima
    messageout = CANPut(TX_Message); //Poner el mensaje CAN en la FIFO de envío
}
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void DIPSWITCH(void)
{
          direccion       = ~PORTD;     //Leer Puerto Dircción
    if (direccion != olddireccion)          //Si la dirección leída es diferente a la anterior
    {
        olddireccion = direccion;     //Dirección anterior es la actual
        if (direccion>7)
        {
            olddireccion = direccion;//Dirección anterior es la actual
            direccion>>=3;
            direccion +=7;
        }
        TX_Message.Address = direccion + 20;     //Sumar 20 a dirección actual
       if(INTCON2bits.INTEDG1 == 0)          //Si modo de escuchar
            Send_CAN_Request();       //Enviar nueva dirección
    }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void REMOTE_MESSAGE(void)
{
    if ((RX_Message.Remote == 1))//&&(RX_Message.Address <20))
    {
        origin       = RX_Message.Address;          //origen es igual al identificador
        origin       = origin<<4;           //poner identificador en 4 bits  Msb
        origin   = origin & 0b11110000;     //borra 4 Lsb de Identificador
        LATB       = (PORTB & 0b00001111)+origin; //Actualizar Dirección Timbre
        Delay10KTCYx(40);//Demorar 40 mil ciclos de Instrucción = 40k*0.1µs = 40ms
        TMR3_Int_Routine();     //Borrar Dirección Timbre
    }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void CANErrorHandler(void)
{
    Delay10KTCYx(100);
    _asm
        RESET
    _endasm
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void BUS_busy(void)
{
    T0CONbits.TMR0ON = 0;//Apagar Timer 0
    TMR0          = timer0delay; //Inicializar Timer 0
    busfree          = 0;//Definir Bus ocupado
    PORTEbits.RE1   = 0;//LED Bus Libre   = 0;
    PORTEbits.RE2   = 1;//LED Bus Ocupado = 1;
    T0CONbits.TMR0ON = 1;//Encender Timer 0
    LATCbits.LATC7       = 1;   //Activar Filtro de Salida
}
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void RECEIVE_PROCEDURE(void)
{
    received = 0;            //Definir que no hay mensajes que descomprimir
    while ((received == 0)  &&(INTCON2bits.INTEDG1 == 0))
    {
        Get_CAN_Message();    //Recoger mensaje CAN
        DIPSWITCH();        //Rutina de Lectura de Identificador
    }
    while ((CAN2bytecount>0)&&(INTCON2bits.INTEDG1 == 0))
    {
        Get_Request();    //Recoger Mensajes Remotos
        DIPSWITCH();   //Rutina de Lectura de Identificador
    }
    if(INTCON2bits.INTEDG1 == 0)
    {
        for(j=0; j<8; j++)
        {
            ADPCM2byte[j] = ReceiveVector[j];//Descargar Vector de Datos Comprimidos
        }                                    //Cuando CANbyte2count = 0
    }

    if ((CANdataempty == 1)&&(INTCON2bits.INTEDG1 == 0))
    {
        CANdataempty = 0;  //Definir que hay mensaje a descomprimir
        TMR1_Int_Routine();    //Encender Timer1
    }                                    //Establece permanencia de las muestras en DAC
}
//*****************************************************************************
                //PROGRAMA PRINCIPAL
//*****************************************************************************
#pragma code mainprogram
void main(void)
{
//CONFIGURACIONES INICIALES
    DEFINE_PORTS();  //Configuración de Puertos
    RB1_CONFIG();      //Configuración de Interrupción Externa INT1
    ADC_CONFIG();     //Configuración de Convertidor A/D
    TIMER0_CONFIG();     //Configuración de Timer0
   TIMER1_CONFIG();      //Configuración de Timer1
    TIMER2_CONFIG();     //Configuración de Timer2
    TIMER3_CONFIG();     //Configuración de Timer3
    SPI_CONFIG();       //Configuración de Puerto SPI (Serial Peripheral Interface)
    CANInit();          //Inicializar Módulo CAN
    RCONbits.IPEN   = 1;      //Existen Niveles de Prioridad de Interrupciones
    INTCONbits.GIE  = 1;   //Habilitar Interrupciones Globales
    INTCONbits.PEIE = 1;   //Habilitar Interrupciones Periféricas

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
    SET_CAN_HEADER();
    while(busfree == 0);  //Mientras el bus esté ocupado
//CICLO PRINCIPAL
    while(1)
    {
        if(INTCON2bits.INTEDG1 == 0)
            RECEIVE_PROCEDURE(); //Rutina de Recepción de Mensajes
        else DIPSWITCH();       //Rutina de Lectura de Identificador
    }
}

//PALABRAS DE CONFIGURACIÓN
//*****************************************************************************
#pragma romdata CONFIG //Configuración Inicial del PIC18F458
_CONFIG_DECL (_OSCS_OFF_1H & _OSC_HSPLL_1H,_PWRT_ON_2L & _BOR_ON_2L &
_BORV_45_2L,
_WDT_OFF_2H & _WDTPS_128_2H, _STVR_ON_4L & _LVP_OFF_4L & _DEBUG_OFF_4L,
_CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L &_CP3_OFF_5L,
_CPB_OFF_5H &_CPD_OFF_5H,
_WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L,
_WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H,
_EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L,
 _EBTRB_OFF_7H);
#pragma romdata
//*****************************************************************************
//FIN CODIGO
```

Listado de la sección de programa en lenguaje C grabado en la memoria de programa del PIC18F258

```
#include  <p18f258.h>
#include "can.h"
#include <delays.h>
#include <adc.h>
//*****************************************************************************
//VARIABLES GLOBALES
const unsigned char sampleperiod =99,spicommand=0b00110000;
const unsigned short timer0delay = 0, timer1delay = 65340;        //40 us = 65340
unsigned short ADCresult;
unsigned char i,j;
unsigned char ADPCMbitcount = 0,CANbytefull = 0;
struct CANMessage RX_Message, TX_Message;
unsigned char CANbytecount = 0, adaptive_step=1, adaptive_sign = 0, SUMADC;
unsigned char infobyte, ADPCMbyte[8], ReceiveVector[8],mensajes=0;
unsigned char CAN2bytecount = 0, adaptive2_step=1, adaptive2_sign = 0, SUM2ADC;
unsigned char info2byte, ADPCM2byte[8], messageout;
unsigned char bitcount, pastbits, received;
unsigned char direccion, olddireccion = 0,busfree = 0, DACcontrol=0, DAC=0;
unsigned char spidata1, spitemp, buzzer = 0;
volatile unsigned char CANdataempty = 1, mychannel=0;
void ADC_Int_Routine(void);//Interrupción del ADC que recoge muestras y al s coloca en un byte CAN
void TMR0_Int_Routine(void); //Define si  el bus CAN está libre luego de 0.4 seg.
void TMR1_Int_Routine(void); //Define el tiempo que una muestra permanece en el DAC
void TMR2_Int_Routine(void); //Acciona el ADC y determina la frecuencia de muestreo (25kHz)
void RB1_Int_Routine(void);    //Establece si el modulo está en modo escuchar o hablar
void Send_CAN_Message(void);    //Coloca un mensaje CAN de voz en la cola de envío
void Send_CAN_Request(void);     //Coloca un mensaje CAN remoto en la cola de envío
void GET_ADPCM_STEP(void);   //Rutina de Compresión de Datos
void PUT_ADC_IN_DAC(void);    //Determina paso a utilizar y si sumarlo o restarlo (Compresor)
void DET_ADPCM_STEP(void);   //Determina paso a utilizar y si sumarlo o restarlo (Descompresor)
void REFRESH_DAC(void);     //Rutina de descompresión de Datos
void SPI_DAC(unsigned char); //Envía serialmente al DAC el dato descomprimido
void SET_CAN_HEADER(void);   //Establece que los mensajes CAN tineen 8 datos y que son standard
//*****************************************************************************
//*****************************************************************************
//LOW PRIORITY INTERRUPT ROUTINES
#pragma   interrupt InterruptHandlerLow  //save=section(".tmpdata")
void InterruptHandlerLow(void)
{
   CANISR();          //CAN INTERRUPT SERVICE ROUTINE
}                                                  //endLowInt
//---------------------------------------------------------------------------
```

```
// HIGH PRIORITY INTERUPT ROUTINES
#pragma interrupt InterruptHandlerHigh
void InterruptHandlerHigh (void)
{
//ADC_INT_FLAG
   if (PIR1bits.ADIF&&PIE1bits.ADIE)            //Interrupt ADC
   {
        ADC_Int_Routine();
        PIR1bits.ADIF = 0;            //Borrar Bandera ADC
   }
//Interrupt Timer2
    else if (PIR1bits.TMR2IF&&PIE1bits.TMR2IE)  //Interrupt Timer2
    {
        TMR2_Int_Routine();
        PIR1bits.TMR2IF = 0;            //Borrar Bandera TMR2
    }
//Interrupt Timer1
    else if (PIR1bits.TMR1IF&&PIE1bits.TMR1IE)  //Interrupt Timer1
    {
        TMR1_Int_Routine();
        PIR1bits.TMR1IF = 0;            //Borrar Bandera TMR1
    }
//Interrupt Timer0
    else if (INTCONbits.TMR0IF)            //Interrupt Timer0
    {
        TMR0_Int_Routine();
        INTCONbits.TMR0IF = 0;       //Borrar Bandera TMR0
    }
//Interrupt RB1
    else if (INTCON3bits.INT1IF)            //Interrupt RB1
   {
        RB1_Int_Routine();
        INTCON3bits.INT1IF = 0;       //Borrar Bandera INT1
   }//end RB1_Int
}//end InterruptHandlerHigh
//------------------------------------------------------------------------------
// Vector de Interrupciones de Baja Prioridad
#pragma code InterruptVectorLow = 0x018
void InterruptVectorLow (void)
{
    _asm
        goto InterruptHandlerLow
    _endasm
}
```

```
//---------------------------------------------------------------------------
// Vector de Interrupciones de Alta Prioridad
#pragma code InterruptVectorHigh = 0x008
void InterruptVectorHigh (void)
{
    _asm
        goto InterruptHandlerHigh
    _endasm
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++/
/INICIO CODIGO
#pragma code
//---------------------------------------------------------------------------
void TMR2_Int_Routine(void)
{
    T2CONbits.TMR2ON   = 0;      //Apagar TIMER2
    ConvertADC();                //Iniciar Conversión AD                TMR2 =  0;                //Recargar
TIMER2
}
//---------------------------------------------------------------------------
void TMR0_Int_Routine(void)
{
    T0CONbits.TMR0ON    = 0;     //Apagar TIMER1
    busfree             = 1;     //Asumir Bus Libre
    LATBbits.LATB7      = 1;     //Encender Luz Verde Bus Libre
    LATBbits.LATB5      = 0;     //Apagar Luz Roja Bus Ocupado
    TMR0                = 0;     //Reiniciar Timer0
    INTCON3bits.INT1IE  = 1;     //Deshabilitar Botón de Hablar
    LATCbits.LATC7      = 0;     //Desactivar Filtro de Salida
}
//---------------------------------------------------------------------------
void TMR1_Int_Routine(void)
{
    T1CONbits.TMR1ON   = 0;          //Apagar TIMER1
    TMR1 = timer1delay;          //Recargar TIMER1
    REFRESH_DAC();         //Descomprimir Dato CAN
    SPI_DAC(DAC);            //Poner Dato Nuevo en DAC
    if (CANdataempty == 0)           //Si no hay mensaje que descomprimir
        T1CONbits.TMR1ON   = 1;   //Encender Timer1 nuevamente
}
//---------------------------------------------------------------------------
void SPI_DAC(unsigned char spidata2)
{
    spitemp    = spidata2>>4;          //MSBits de DATO van en LSBits de 1er byte SPI
    spitemp       = spitemp&0b00001111;   //Borrar MSBits
    spidata1  += spitemp;          //1er byte lleva MSB del dato y Palabra de Config.
    LATAbits.LATA5    = 0;          //CS = 0    //Chip Select DAC = 0
    SSPBUF      = spidata1; //0010-1111//Cargar Buffer SPI con 1er. byte SPI
    spidata2    = spidata2<<4;          //Poner dato DAC en MSBits de 2do. byte SPI
    Delay1TCY();//1           //Esperar  un Ciclo de Reloj = NOP
```

```
    spidata1   = spicommand;          //Recargar Palabra Configuración en 1er. byte
    SSPBUF     = spidata2;        //10101010//Cargar Buffer SPI con 2do. byte SPI
    Delay1TCY();               //NOP 1
    Delay1TCY();               //NOP 2
    Delay1TCY();               //NOP 3
    Delay1TCY();               //NOP 4
    Delay1TCY();               //NOP 5
    Delay1TCY();               //NOP 6
    Delay1TCY();               //NOP 7
    Delay1TCY();               //NOP 8
    LATAbits.LATA5    = 1;          //Chip Select DAC = 1
}
//------------------------------------------------------------------------------
void RB1_Int_Routine(void)
{
    if ((INTCON2bits.INTEDG1 == 0))//Interrupt cayó en flanco de bajada
    {
        if ((mychannel == 1)&&(busfree == 1)) //Si el canal está habilitado y bus libre
        {
            INTCON2bits.INTEDG1 = 1;    //Detectar Flanco Subida
            LATBbits.LATB6      = 1;          //Activar Luz Amarilla
            LATCbits.LATC6      = 1;          //Activar Filtro de Entrada

            T1CONbits.TMR1ON    = 0;          //Apagar TIMER1
            PIE1bits.TMR1IE     = 0;          //Deshabilitar Interrupt TIMER1
            PIR1bits.TMR1IF     = 0;          //Borrar Bandera TIMER1
            CANbytecount        = 0;       //Borrar Contador de byte a comprimir
            PIE1bits.ADIE       = 1;       //Habilitar Interrupt ADC
            PIE1bits.TMR2IE     = 1;       //Habilitar Interrupt TIMER2
            SET_CAN_HEADER();      //Establecer Parametros de Envío CAN
            TMR2_Int_Routine();          //Encender TIMER2
        }
        else                        //si el canal deshabilitado o bus ocupado
        {
            INTCON2bits.INTEDG1 = 1;    //Detectar Flanco de Bajada
            LATBbits.LATB6 = 1;         //Encender LED Hablar
            T1CONbits.TMR1ON    = 0;          //Apagar TIMER1
            PIE1bits.TMR1IE     = 0;          //Deshabilitar Interrupt TIMER1
            PIR1bits.TMR1IF     = 0;          //Borrar Bandera TIMER1
            buzzer              = 0;          //Mensaje Remoto aún no enviado
        }
    }
    else if(INTCON2bits.INTEDG1 == 1)  //Interrupt cayó en flanco de subida
    {
        INTCON2bits.INTEDG1 = 0;    //Detectar Flanco de Bajada
        T2CONbits.TMR2ON    = 0;       //Apagar TIMER2
        PIE1bits.TMR2IE     = 0;     //Deshabilitar Interrupt TIMER2
        PIR1bits.TMR2IF     = 0;     //Borrar Bandera TIMER2
        ADCON0bits.GO_DONE  = 0;  //Detener Conversión AD
        PIE1bits.ADIE       = 0;     //Deshabilitar Interrupt ADC
```

```
        PIR1bits.ADIF       = 0;    //Borrar Bandera ADC
        PIE1bits.TMR1IE     = 1;    //Habilitar Interrupt TIMER1
        CANdataempty        = 1;    //Definir que no hay datos para descomprimir
        CAN2bytecount       = 0;    //Borrar Contador de byte a descomprimir
        LATBbits.LATB6          = 0;    //Desactivar Luz Amarilla
        LATCbits.LATC6      = 0;    //Desactivar Filtro de Entrada
    }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void ADC_Int_Routine(void)
{
    T2CONbits.TMR2ON   = 1; //Encender TIMER2
    ADCresult = ReadADC();          //Leer Resultado AD (16bits)
    ADCresult>>=8;              //Correr Resultado 8 posiciones hacia LSb
    if (CANbytecount > 0)
    {
        infobyte = infobyte>>1;//Correr 1 pos en byte de Resultados  hacia LSb
        if (ADCresult > DACcontrol) //El cambio se dio en dirección de aumento
        {
            infobyte+=128;//Si Resultado AD>Resultado Adaptativo bit comprimido = 1
            adaptive_sign = 1;//Incrementar puntero de byte de bits/datos comprimidos
        }//endif
        else adaptive_sign = 0;   //Si Resultado Adaptativo<Resultado AD bitl comprimido = 0
        ADPCMbitcount++;  //Incrementar puntero de byte de bits/datos comprimidos
        if (ADPCMbitcount == 8)
        {
            CANbytefull   = 1;//Byte Comprimido está lleno si ya se juntaron 8 bits
    ADPCMbitcount = 0;//Borrar ountero de byte de bits/datos comprimidos
        }
    }
    else  //CANDATA = ADCRESULT
    {
        infobyte = ADCresult;//Si es el primer resultado A/D ponerlo directamente en mensaje
    CANbytefull = 1; //Definir que el byte actual está lleno
    }//endelse
//CARGAR BYTE EN CANBUFFER
        if (CANbytefull == 1)         //Si el byte actual está lleno
        {
            ADPCMbyte[CANbytecount]= infobyte;//Colocar byte lleno en mensaje CAN
            CANbytecount++;         //Incrementar contador de bytes en mensaje
            if (CANbytecount > 7)       //Si el contador de mensajes es amyor a 7
            {
                Send_CAN_Message();  //Enviar mensaje con 8 bytes
                CANbytecount = 0;   //Reiniciar contador de mensajes
            }                                                   CANbytefull   = 0;    //Definir que byte actual de
datos está vacío
            ADPCMbitcount = 0;     //Reiniciar contador de bits en byte comprimido
        }                                               PUT_ADC_IN_DAC();       //Simular que se pone un
dato en el DAC
}
```

```
//---------------------------------------------------------------------------
void Send_CAN_Message(void)
{
    for (i=0;i<8;i++)  //Cargar Mensaje CAN de 8 bytes en otro vector
    {
        TX_Message.Data[i] = ADPCMbyte[i];
    }
    messageout = CANPut(TX_Message);//Poner mensaje en  FIFO de Envío CAN
}
//---------------------------------------------------------------------------
void GET_ADPCM_STEP(void)
{
    SUMADC = (infobyte&128) + (infobyte&64)+ (infobyte&32)+(infobyte&16);
    if ((SUMADC==240)||(SUMADC==0))     adaptive_step *= 2;   //Duplicar paso anterior
    else                        adaptive_step  = 1;    //Regresar a paso =  1
    if (adaptive_step>=64)              adaptive_step = 32;   //Usar paso máximo
    else if (adaptive_step<1)      adaptive_step =  1; //Usar paso mínimo
}
//---------------------------------------------------------------------------
void PUT_ADC_IN_DAC(void)
{
    if ((CANbytecount ==1)&&(ADPCMbitcount==0))//Si es el primer byte de los 8 a enviar
    {
        DACcontrol = infobyte;       //poner byte completo de conversion en DAC
        adaptive_step = 1;          //Regresar paso a unitario
        infobyte = 42;           //Forzar serie de valores anteriores a 10101010
    }
    else
    {
        GET_ADPCM_STEP();       //Definir paso a utilizar
        if (adaptive_sign == 1)         //Si es un aumento sumar paso delta
        {
            if ( DACcontrol <(0XFF-adaptive_step)) DACcontrol += adaptive_step;
            else DACcontrol = 0XFF;         //Saturar hacia arriba
        }
        else                    //Si es un disminución restar paso delta
        {
            if (DACcontrol > adaptive_step) DACcontrol -= adaptive_step;
            else DACcontrol = 0X00;         //Saturar hacia abajo
        }
    }
}
```

```
//---------------------------------------------------------------------------
void DET_ADPCM_STEP(void)
{
    SUM2ADC = (pastbits&128) + (pastbits&64) + (pastbits&32)+(pastbits&16);
    if ((SUM2ADC == 240 )||(SUM2ADC == 0)) adaptive2_step<<=  1;//Duplicar paso
    else                        adaptive2_step  =  1;//Volver Paso Unitario
    if     (adaptive2_step >= 64)       adaptive2_step  = 32;//Usar paso máximo
    else if (adaptive2_step <   1)       adaptive2_step  =  1;//Usar paso mínimo
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void REFRESH_DAC(void)
{
    if (CAN2bytecount == 0)             //Si es el primer byte del mensaje
    {
        DAC = ADPCM2byte[CAN2bytecount];//Estado DAC es actualizado con primer byte
    pastbits = 42;          //Trama anterior forzada a ser 10101010
        adaptive2_step = 1;       //Reducir paso delta a la unidad
        CAN2bytecount++;       //Aumentar contador de bytes CAN
        info2byte = ADPCM2byte[CAN2bytecount];//Recoger siguiente byte de datos del buffer
    bitcount = 1;               //Reiniciar contador de muestras de byte
    }
    else
    {
        pastbits>>=1;          //Mover bits de muestra anteriores una posición
        adaptive2_sign = info2byte & bitcount;//Obtener el bit de muestra actual
        if (adaptive2_sign > 0)
        {
            adaptive2_sign = 1;   //Bit de muestra actual es uno
            pastbits +=128;       //Actualizar byte muestras anteriores con bit actual
        }
        else adaptive2_sign = 0;      //Bit de muestra actual es cero
        DET_ADPCM_STEP();      //Determinar paso delta a utilzar
        if (adaptive2_sign == 1)
        {
            if ( DAC <(0XFF - adaptive2_step)) DAC += adaptive2_step; //Sumar paso                    else
DAC = 0XFF;       //Saturar hacia arriba
        }
        else
        {
            if (DAC > adaptive2_step) DAC -= adaptive2_step;    //Restar paso delta
            else DAC = 0X00;     //Saturar hacia abajo
        }
        bitcount<<=1;          //Duplicar Contador de muestras en byte
        if (bitcount == 0)      //Si contador de bits llegó al final, reiniciarlo
        {
            bitcount = 1;  //Reiniciar Contador de muestras en byte
            CAN2bytecount++;//Aumentar contador de byte CAN a leer
```

```
            if (CAN2bytecount > 7)//Si  contador mayor que 7 reiniciar variables
            {
                CAN2bytecount = 0;//Reiniciar contador de byte CAN a leer
                CANdataempty  = 1;//Definir que ya no hay mas datos a descomprimir
            }
            else info2byte = ADPCM2byte[CAN2bytecount]; //Actualizar byte de muestras
        }
    }
}//endroutine
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void ADC_CONFIG( void )
{
// configure A/D convertor
    OpenADC( ADC_FOSC_64 & ADC_LEFT_JUST & ADC_1ANA_2REF, ADC_CH0 &
ADC_INT_ON );
    ADCON0bits.GO_DONE = 0;     //Detener Conversión AD
    PIE1bits.ADIE      = 0;     //Deshabilitar Interrupción por ADC
    PIR1bits.ADIF      = 0;     //Borrar Bandera ADC
    IPR1bits.ADIP      = 1;   //Interrupción AD es de alta prioridad
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void RB1_CONFIG(void)
{
    //RB1_INT
    TRISBbits.TRISB1    = 1;    //Puerto RB1 es un entrada digital
    INTCON2bits.RBPU = 0;    //Encender Pull-Ups de Puerto B
    INTCON2bits.INTEDG1  = 0;   //Interrupción INT1 activada con flanco de caída
    INTCON3bits.INT1IP   = 1; //Interrupción INT1 es de alta prioridad
    INTCON3bits.INT1IF   = 0; //Borrar bandera de Interrupción INT1
    INTCON3bits.INT1IE   = 1; //Habilitar Interrupción externa INT1
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TIMER0_CONFIG (void)
{
//OpenTimer0( TIMER_INT_ON & T0_8BIT & T0_SOURCE_INT & T0_PS_1_256 );
//config T0CON
    T0CONbits.TMR0ON   = 0; //Apagar Timer 0
    T0CONbits.T08BIT   = 0;         //Timer 0 es un 16 bit timer/counter
    T0CONbits.T0CS     = 0;         //Fuente de Reloj es interna Fosc/4
    T0CONbits.PSA      = 0;      //Asignar Prescaler
    T0CONbits.T0PS0    = 1;         //Prescaler Select 0.1µ*64*65535 = 0.4194 s.
    T0CONbits.T0PS1    = 0;         //Prescale Value = 1:64 (0.4 seg.)
    T0CONbits.T0PS2    = 1;
//enable TMR0 Overflow Interrupt
    INTCON2bits.TMR0IP = 1; //Timer0 tieneInterrupción de Alta Prioridad
    INTCONbits.TMR0IF  = 0;  //Borrar bandera de Interrupción Timer 0
    INTCONbits.TMR0IE  = 1; //Habilitar Interrupción de Timer 0
```

```
//reset TMR0
    TMR0 = timer0delay;          //Cargar Valor Inicial de Timer 0
    T0CONbits.TMR0ON   = 1;      //Encender Timer 0;
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TIMER1_CONFIG (void)
{
//config T0CON
    T1CONbits.RD16     = 1;              //Habilitar 16 bit read/write
    T1CONbits.T1CKPS0  = 0;      //Prescale Value = 1:1
    T1CONbits.T1CKPS1  = 0;
    T1CONbits.T1OSCEN  = 0;      //Opción de Oscilador externo deshabilitada
    T1CONbits.T1SYNC   = 1;
    T1CONbits.TMR1CS   = 0;      //Fuente de Reloj es interna Fosc/4
//enable TMR1 Overflow Interrupt
    IPR1bits.TMR1IP    = 1;          //Interrupción de Timer1 es de Alta Prioridad
    PIR1bits.TMR1IF    = 0;          //Borrar Bandera de Interrupción Timer1
    PIE1bits.TMR1IE    = 1;          //Interrupción de Timer1  habilitada
//reset TMR1                  //65535-6340 = 195
    TMR1 = timer1delay;              //195*0.1µs = 19.5µs
    T1CONbits.TMR1ON   = 0;        //Apagar Timer1
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void TIMER2_CONFIG (void)
{
//POSTSCALER
    T2CONbits.TOUTPS0  = 1;      //Prescaler = 1:2
    T2CONbits.TOUTPS1  = 0;      //Timer2 se desborda cada 98*2*0.1µs =19.6µs
    T2CONbits.TOUTPS2  = 0;
    T2CONbits.TOUTPS3  = 0;
//PRESSCALER
    T2CONbits.T2CKPS0  = 0;
    T2CONbits.T2CKPS1  = 0;      //Postscaler = 1:1
//PERIOD
    PR2 = sampleperiod;          //Sampleperiod  = 98
//INTERRUPT
    PIR1bits.TMR2IF   = 0;           //Borrar Bandera de Interrupción Timer 2
    PIE1bits.TMR2IE   = 1;           //Deshabilitar Interrupción por Timer 2
    IPR1bits.TMR2IP    = 1;          //Timer2 is High Priority
//TMR2 IS OFF
    TMR2 = 0;                  //Inicializar Timer 2
    T2CONbits.TMR2ON  = 0;       //Apagar Timer 2
}
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void DEFINE_PORTS(void)
{
    TRISC = 0;              //Puerto C son Salidas
    TRISAbits.TRISA0 = 1;      //Canal 0
    TRISAbits.TRISA2 = 1;      //RA2 es Vref -
    TRISAbits.TRISA3 = 1;      //RA3 es Vref +
    TRISBbits.TRISB7 = 0;      //RB7    = LED Bus Libre
    TRISBbits.TRISB6 = 0;      //RB6    = LED Hablando
    TRISBbits.TRISB5 = 0;      //RB5    = LED Bus Ocupado
    LATBbits.LATB7     = 0;   //Apagar LED Bus Libre
    LATBbits.LATB6     = 0;   //Apagar LED Hablando
    LATBbits.LATB5     = 1;   //Apagar LED Bus Ocupado
    TRISCbits.TRISC6 = 0;      //Salida Activadora Filtro de Salida
    TRISCbits.TRISC7 = 0;      //Salida Activadora Filtro de Entrada
    LATCbits.LATC6     = 0;   //Filtro de Entrada  Inactivo
    LATCbits.LATC7     = 0;   //Filtro de Salida   Inactivo
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void SPI_CONFIG(void)
{
//SPI PORTS
    TRISAbits.TRISA5  = 0;          //CHIP SELECT del SPI (SS)
    TRISCbits.TRISC3  = 0;          //SPI_CLK - Reloj del SPI
    TRISCbits.TRISC5  = 0;          //SPI_SDOUT - Salida Serial SPI
//SPI Iterrupt
    PIE1bits.SSPIE     = 0;        //Deshabilitar Interrupción SPI
    PIR1bits.SSPIF     = 0;        //Borrar Bandera de Interrrupción SPI
    IPR1bits.SSPIP    = 0;         //Establecer Interrupción SPI como de Alta prioridad
//STATUS REGISTER
    SSPSTATbits.SMP     = 0;   //Muestrear Datos a la mitad del bit
    SSPSTATbits.CKE     = 0;   //Transmitir Datos en el flanco de subida
//CONTROL REGISTER
    SSPCON1bits.WCOL  = 0;  //Borrar Bit de Estado Colisiones
    SSPCON1bits.SSPOV = 0;  //Borrar Bit de Desborde
//SPI MODE: Fosc/4 = 10 MHz
    SSPCON1bits.SSPM0 = 0;  //Modo de Operación del SPI es maestro a 10bits/s.
    SSPCON1bits.SSPM1 = 0;
    SSPCON1bits.SSPM2 = 0;
    SSPCON1bits.SSPM3 = 0;
//CONFIGURE SDI, SDO, CLK, CS
    SSPCON1bits.CKP   = 1;          //Estado pasivo de línea de reloj es un uno lógico
    SSPCON1bits.SSPEN = 1;          //Encender módulo SPI
}
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void SET_CAN_HEADER(void)
{
                TX_Message.Address = MY_ADDRESS_IDENTIFIER+20;//Establecer el identidficador
                del mensaje
        #ifdef MY_ADDRESS_IS_STANDARD
        TX_Message.Ext       = 0;    //Si el identificador es standard, Ext. = 0
        #else
        TX_Message.Ext       = 1;    //Si el identificador es extendido, Ext. = 1
        #endif
        TX_Message.NoOfBytes = 8;//Definir número de bytes a enviar
        TX_Message.Remote    = 0;//Borrar Bandera de Mensaje Remoto
                TX_Message.Priority  = 0;//Prioridades FIFO de CAN  0->mínima prioridad, 3->máxima
}




//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Send_CAN_Request(void)
{
    TX_Message.Address      = MY_ADDRESS_IDENTIFIER;
    TX_Message.Ext          = 0;    //Identificador es estándar
    TX_Message.NoOfBytes = 0;       //Numero de bytes es cero
    TX_Message.Remote       = 1;        //Activar bandera de mensaje remoto
    TX_Message.Priority  = 3;       //Prioridad FIFO 0-> mínima, 3-> máxima
    messageout = CANPut(TX_Message);//Poner el mensaje CAN en la FIFO de envío
}


//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void Get_CAN_Message(void)
{
    if(CANRXMessageIsPending())//Check if there is an unread CAN message
    {
        RX_Message = CANGet();                          //Get the message
        if((RX_Message.Remote == 0)&& (RX_Message.Address ==
           MY_ADDRESS_IDENTIFIER+20))
        {
            for(j=0; j<RX_Message.NoOfBytes; j++)  //Fill the Data bytes
            {
                ReceiveVector[j] = RX_Message.Data[j] ;
            }
           received       = 1;
            mychannel       = 1;
            LATCbits.LATC2    = 1;      //channel  is mine
            INTCON3bits.INT1IE = 0;
            INTCON3bits.INT1IF = 0;
        }
```

```
                else if ((RX_Message.Remote == 1)&&(RX_Message.Address ==
                MY_ADDRESS_IDENTIFIER+20))
        {
            mychannel     = 1;
            LATCbits.LATC2 = 1;   //channel occupied
        }
        else if ((RX_Message.Remote == 1)&&(RX_Message.Address>=20))
        {
            mychannel     = 0;
            LATCbits.LATC2 = 0;   //channel occupied
        }
    }
}//endroutine
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void CANErrorHandler(void)
{
    Delay10KTCYx(100);
    _asm
        RESET
    _endasm
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void BUS_busy(void)
{
    T0CONbits.TMR0ON = 0;  //Apagar Timer 0
    TMR0              = timer0delay;//Inicializar Timer 0
    busfree         = 0;//Definir Bus ocupado
    PORTBbits.RB7   = 0;    //LED Bus Libre   = 0;
    PORTBbits.RB5   = 1;    //LED Bus Ocupado = 1;
    T0CONbits.TMR0ON = 1;  //Encender Timer 0
    if (mychannel == 1)       LATCbits.LATC7     = 1;     //Activar    Filtro de Salida
    else                      LATCbits.LATC7     = 0;     //Desactivar Filtro de Salida
}
```

```
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void RECEIVE_PROCEDURE(void)
{
    received = 0;            //Definir que no hay mensajes que descomprimir
    while ((received == 0)&&(INTCON2bits.INTEDG1 == 0))Get_CAN_Message();
          //Recoger mensaje CAN
    while((CAN2bytecount>0)&&(INTCON2bits.INTEDG1 == 0));
    if (INTCON2bits.INTEDG1 == 0 )
    {
        for(j=0; j<8; j++)
        {
            ADPCM2byte[j] = ReceiveVector[j];//Descargar Vector de Datos Comprimidos
        }                                    //Cuando CANbyte2count = 0
    }



    if ((CANdataempty == 1)&&(INTCON2bits.INTEDG1 == 0 ))
    {
        CANdataempty = 0;       //Definir que hay mensaje a descomprimir
        TMR1_Int_Routine();         //Encender Timer1
    }                                    //Establece permanencia de las muestras en DAC
}
```

```
//*****************************************************************************
                //PROGRAMA PRINCIPAL
//*****************************************************************************
#pragma code mainprogram
void main(void)
{
//CONFIGURACIONES INICIALES
    DEFINE_PORTS();  //Configuración de Puertos
    RB1_CONFIG();      //Configuración de Interrupción Externa INT1
    ADC_CONFIG();     //Configuración de Convertidor A/D
    TIMER0_CONFIG();     //Configuración de Timer0
   TIMER1_CONFIG();       //Configuración de Timer1
    TIMER2_CONFIG();     //Configuración de Timer2
    SPI_CONFIG();       //Configuración de Puerto SPI (Serial Peripheral Interface)
    CANInit();          //Inicializar Módulo CAN
    RCONbits.IPEN   = 1;         //Existen Niveles de Prioridad de Interrupciones
    INTCONbits.GIE  = 1;   //Habilitar Interrupciones Globales
    INTCONbits.PEIE = 1;   //Habilitar Interrupciones Periféricas
////////////////////////////////////////////////////////////////////////////////////////////
    SET_CAN_HEADER();
//CICLO PRINCIPAL
    while (busfree==0);  //Mientras el bus esté ocupado
    while(1)
    {
        if (INTCON2bits.INTEDG1==0)RECEIVE_PROCEDURE();
                //Rutina de Recepción de Mensajes
        else
        {
            if((mychannel == 0)&&(PIE1bits.ADIE==0)) //Si canal deshabilitado
            {
                if (buzzer == 0)
                {
                    Send_CAN_Request();   //Enviar Mensaje Remoto
                    buzzer = 1;          //Mensaje Remoto Enviado
                    }
                }
            }
    }
}
```

```
//PALABRAS DE CONFIGURACIÓN
//*****************************************************************************
#pragma romdata CONFIG  //Configuración Inicial del PIC18F258
_CONFIG_DECL (_OSCS_OFF_1H & _OSC_HSPLL_1H,_PWRT_ON_2L & _BOR_ON_2L &
_BORV_45_2L,
_WDT_OFF_2H & _WDTPS_128_2H, _STVR_ON_4L & _LVP_OFF_4L & _DEBUG_OFF_4L,
_CP0_OFF_5L & _CP1_OFF_5L & _CP2_OFF_5L &_CP3_OFF_5L,
_CPB_OFF_5H &_CPD_OFF_5H,
_WRT0_OFF_6L & _WRT1_OFF_6L & _WRT2_OFF_6L & _WRT3_OFF_6L,
_WRTC_OFF_6H & _WRTB_OFF_6H & _WRTD_OFF_6H,
_EBTR0_OFF_7L & _EBTR1_OFF_7L & _EBTR2_OFF_7L & _EBTR3_OFF_7L,
 _EBTRB_OFF_7H);
#pragma romdata
//*****************************************************************************
//FIN CODIGO
```