TRANSMISIÓN POR PUERTO SERIE UTILIZANDO COMPRESIÓN DE DATOS

JAIME TEJEDOR GÓMEZ

MEMORIA DEL PROYECTO DE FIN DE CARRERA

TUTOR: ALFREDO PÉREZ VEGA-LEAL

ÍNDICE

1. OBJETIVOS, MOTIVACIÓN

El objetivo del proyecto en un principio consistía en crear un algoritmo de compresión que se ejecutase en una placa externa utilizando el microprocesador o DSP de la placa, de tal forma que al transmitir los datos sin comprimir desde el PC, estos se comprimiesen en la placa antes de ser transmitidos. Los datos transmitidos por la placa llegarían hasta otra placa situada en el otro extremo de la comunicación, de tal forma que la segunda placa descomprimiría los datos y se los pasaría descomprimidos al PC de destino.

El principal objetivo consistía en implementar el algoritmo de compresión fuera del PC, de tal forma que no se consumiese tiempo de CPU en el PC. Esto no tiene demasiada utilidad práctica, ya que hoy en día los procesadores son cada vez mas potentes, y el tiempo de compresión es mínimo respecto al tiempo que se tarda en enviar los datos, sin contar con que realizar el desarrollo hardware es mas caro, y además, si la velocidad de comunicación entre la placa y el PC está limitada, esto puede convertirse en un cuello de botella, pero el concepto de compresión por hardware si puede ser útil en tarjetas de comunicación, modems y otros dispositivos que puedan comunicarse comprimiendo los datos en tiempo real sin necesidad de tener un PC para que les haga la tarea (lo cual puede ser inviable o resultar muy caro en determinadas situaciones), como pueda ser el caso de aparatos de medición remotos que en lugar de estar transmitiendo datos todo el tiempo, los almacene en un buffer y cada cierto tiempo los transmita comprimidos.

Por lo tanto el objetivo original cambió, pasando a ser el diseño de una placa como la pensada inicialmente y la implementación de un algoritmo diseñado según las características peculiares de la placa, tales como la baja cantidad de memoria disponible y la necesidad de ser un algoritmo lo mas rápido posible, al requerirse compresión en tiempo real. La implementación del algoritmo se codifica para PC, empleandose en un programa que permita el intercambio de archivos, a los que se le aplica el codec de compresión que se ha diseñado a medida de la placa.

Por otra parte, la placa diseñada se ha realizado físicamente para constatar los posibles problemas de diseño, lo cual ha dado lugar a un nuevo esquema ligeramente modificado para tratar de minimizar estos problemas.

La comunicación de los ordenadores entre si se realiza mediante puerto serie, dada su sencillez de programación, bajo coste y alta disponibilidad.

2. MEMORIA DESCRIPTIVA

2.1 LA TRANSMISIÓN POR PUERTO SERIE

2.1.1. Que son las comunicaciones serie

Cuando nos comunicamos en serie, cada byte o carácter de datos que mandamos o recibimos se envía bit a bit. Cada uno de estos bits puede estar en On o en Off, aunque a veces estos estados se conocen como marca (mark) para el estado On y espacio (space) para el estado Off.

La velocidad de la transmisión de datos en serie se suele expresar a menudo como bits por segundo (bps). Esto representa sencillamente el número de unos y ceros que pueden ser enviados en un segundo. 9600 y 19200 son velocidades comunes que se utilizaban en el interfaz RS-232. Muchas veces la velocidad se mide en kbps, así las velocidades anteriores se representan por 9.6Kbps, 19.2Kbps. Los bits por segundo no deben confundirse con los baudios, que a veces pueden expresar lo mismo pero otras no, ya que los baudios representan el número de cambios en la señal modulada por segundo, y si la modulación emplea 4 símbolos, los bps serán el doble del valor de la velocidad en baudios.

Cuando nos referimos a dispositivos o puertos serie, se etiquetan como Equipo de Comunicaciones de Datos (Data Communications Equipment (DCE)), o Equipo Terminal de Datos (Data Terminal Equipment (DTE)). La diferencia entre ellos es muy simple. Cada par de señales, tales como transmitir y recibir, están intercambiadas. Cuando se conectan 2 DTEs o 2 DCEs juntos, debe emplearse un cable de modem nulo (null-modem) o un adaptador para que intercambie las señales emparejadas.

2.1.2. El estándar RS-232

RS-232 es un estandar de interfaz eléctrico para comunicación de datos en serie definido por la Electronic Industries Association (EIA). Existen 3 versiones diferentes, cada una definiendo un rango diferente de tensiones para los niveles On y Off. La variedad mas utilizada es la RS-232C, para la cual una tensión entre –3V y –12V define un bit "marca", y una tensión entre +3V y +12V define un bit "espacio". La especificación RS-232C dice que estas señales pueden llegar hasta 8 metros antes de que pierdan su efectividad.

RS-232 es también uno de los interfaces de ordenador mas populares de todos los tiempos. Está incluido prácticamente en todos los PCs al mismo tiempo que en muchos otros tipos de ordenadores, desde microcontroladores a mainframes, y los dispositivos que conecta. El uso mas común del interfaz RS-232 es la conexión con un modem, pero otros circuitos con intefaces RS-232 incluyen impresoras, módulos de adquisición de datos, instrumentos de prueba y circuitos de control. También podemos utilizar RS-232 como un enlace simple entre ordenadores de cualquier tipo.

Hoy día, existen interfaces que son mas rápidos y sofisticados, pero RS-232 sigue siendo popular debido a que los requerimientos de hardware y programación son sencillos y baratos, y por que hay un gran número de dispositivos que vienen equipados con este interfaz. Otras elecciones incluyen descendientes del RS-232 que son mas rápidos o baratos, pero conservan la compatibilidad con RS-232 en muchos aspectos.

Otros estándares que podemos encontrar para interfaces serie son el RS-422, que usa tensiones mas bajas y señales diferenciales para permitir longitudes de cable de hasta 300 metros aproximadamente y el RS-574, que define el conector estándar para PC de 9 pines y sus tensiones.

2.1.2.1. Ventajas

- Está ampliamente extendido. Todos los PCs tienen uno o mas puertos RS-232. Los ordenadores nuevos soportan ahora otros interfaces serie como el USB, pero RS-232 puede hacer cosas que USB no puede.
- Al usar microcontroladores, los chips de interfaz hacen muy sencilla la conversión de la tensión de alimentación a las que necesita la transmisión serie.
- Los enlaces pueden ser de hasta 30 metros. La mayor parte de los periféricos no están diseñados para colocarse demasiado lejos. Los enlaces USB pueden llegar hasta los 5 metros, y el puerto paralelo del PC puede llegar alcanzar entre 3 y 5 metros.
- Tan solo se necesitan 3 cables para una comunicación en los 2 sentidos. Un enlace en paralelo tiene típicamente 8 líneas de datos, dos o mas señales de control y varios cables de tierra. El coste de todos los cables y conectores mas grandes se paga.

2.1.2.2. Desventajas

- Si el otro lado del enlace requiere datos en paralelo, necesitamos alguna forma de convertir los datos entre serie y paralelo. De cualquier forma, esto es sencillo utilizando una UART (Universal Asynchronous Reciever-Transmiter).
- Los puertos paralelos son tan útiles que puede ser difícil encontrar un puerto que esté libre para ser usado. Los PCs pueden tener varios puertos, pero un sistema es posible que no disponga de una petición de interrupción dedicada para cada uno. La mayor parte de los microcontroladores solo disponen de un puerto serie en hardware.
- No puede haber mas de dos dispositivos en un enlace.
- La máxima velocidad de transmisión especificada es de 20000 bits por segundo, aunque muchos chips pueden exceder esta cifra, especialmente en distancias mas cortas.
- Los enlaces muy largos requieren otro interfaz.

2.1.2.3. Señales

El estándar RS-232 define 3 cosas sobre la interfaz: Los nombres y funciones de las señales en el enlace, las características eléctricas de las señales y aspectos mecánicos, incluyendo la asignación de pines.

Aunque el estandar designa 25 líneas en el interfaz, los PCs y muchos otros dispositivos raramente tienen mayor soporte que las 9 señales principales que aparecen en la tabla 2.1.1. Las señales adicionales están pensadas para usarse con modems síncronos, canales de transmisión secundarios y selección de la velocidad de transmisión en modems de velocidad dual, los cuales no son comunes hoy en día.

Pin	Señal	Fuente	Tipo	Descripción
1	CD	DCE	Control	Carrier Detect (Detección de portadora)
2	RD	DCE	Datos	Received Data (Datos recibidos)
3	TD	DTE	Datos	Transmitted Data (Datos transmitidos)
4	DTR	DTE	Control	Data Terminal Ready (Terminal de Datos Listo)
5	GND	-	Referencia	Ground (Tierra)
6	DSR	DCE	Control	Data Set Ready (Datos Listos)
7	RTS	DTE	Control	Request To Send (Petición de envío)
8	CTS	DCE	Control	Clear To Send (Listo para Enviar)
9	RI	DCE	Control	Ring Indicator (Indicador de Llamada)

Tabla 2.1.1. – Las 9 señales mas utilizadas del estándar RS-232.

La tabla 2.1.1. utiliza nombres de las señales que son básicamente abreviaturas de las funciones de las señales. Mucha terminología del estándar RS-232 refleja su origen como estandar para la comunicación entre un terminal de ordenador y un modem. Un terminal “tonto” es poco mas que un teclado, una pantalla y puertos de comunicación para acceder a un ordenador remoto. Un enlace RS-232 conecta el terminal al modem, que se conecta a la línea telefónica usada para contactar con el ordenador remoto. Los terminales “listos” tienen alguna inteligencia, pero carecen de almacenamiento en disco o alguna otra característica de un ordenador de sobremesa completo. Con software de emulación de terminal, como por ejemplo el Hyperterminal de Windows, los PCs pueden emular muchos tipos de terminales de ordenador.

Hoy en día, por supuesto, RS-232 hace mucho mas que la comunicación entre un modem y un terminal, en lugar de un terminal suele encontrarse un ordenador completo, aunque también podría tratarse de un pequeño microcontrolador. En lugar de un modem, en el otro lado del enlace podemos conectar un ratón, una impresora, otro PC o microcontrolador o cualquier otra cosa que podamos imaginar.

El estandar llama al terminal final del enlace el Data Terminal Equipment (Equipo Terminal de Datos), abreviado como DTE, y al modem Data Communications Equipment (Equipo de Comunicación de Datos), abreviado como DCE.

No importa que dispositivo del enlace sea el DTE y cual el DCE, pero un enlace debe tener uno de cada. El tipo determina que señales son entradas y cuales son salidas en el conector.

Todos las señales se nombran desde la perspectiva del DTE. Por ejemplo, TD (transmisión de datos) es una salida en un DTE y entrada en un DCE.

Con pocas excepciones, los puertos serie de los PCs se configuran como DTEs, y todos los puertos serie de los modems como DCEs, al igual que la mayoría de los periféricos.

2.1.2.4. Las 9 señales del PC

Las tres señales esenciales para la comunicación bidireccional son:
TD: Transporta datos desde el DTE al DCE. También conocida como TX o TXD.
RD: Transporta datos desde el DCE al DTE. También conocida como RX o RXD.
GND: Referencia de tierra.

Las otras señales se utilizan como controles opcionales para comunicar acerca de la disponibilidad de un dispositivo, o la presencia de una señal portadora o de llamada en una línea telefónica.

Hay dos pares de señales de protocolo handshaking: DTR/DSR y RTS/CTS. Cada pareja tiene sus usos definidos por el estándar.

Hay varias formas de describir el estado de RS-232 y otras señales de control. Una señal con una tensión válida positiva puede ser descrita como On, encendida o verdadera (True), para indicar que está en estado activo. Por ejemplo, cuando DTR es True, el terminal de datos está listo. Para traer la señal True, el dispositivo controlador se dice que sube la línea. Una señal con una tensión válida negativa puede ser descrita como Off, apagada, o Falsa, para indicar que está en estado inactivo. Por ejemplo, cuando DTR no es False, el terminal de datos no está listo. Para traer la señal False, el dispositivo de control se dice que baja la línea.

El protocolo DTR/DSR está orientado a proveer información sobre el estado de la línea telefónica u otro canal de comunicaciones conectado al modem. El terminal levanta DTR para pedirle al modem que se conecte al canal de comunicaciones. En respuesta, el modem sube DSR para indicar que se ha conectado. DSR es falso cuando el modem no está conectado al canal de comunicaciones (o al detectar la desconexión, por ejemplo), o al detectar una falta.

El terminal también puede subir DTR en respuesta a RI, para pedirle al modem que conteste la llamada entrante. En algunos enlaces, DTR y DSR se suben al encederse y tan solo indican que el equipo está presente y alimentado.

El protocolo RTS/CTS provee información adicional sobre si un dispositivo está listo o no para recibir datos. Hay dos usos comunes para estas señales.

El primer y original uso, la pareja provee un protocolo handshaking completo. Cuando el terminal tiene datos para enviar, sube RTS. En respuesta, el modem sube CTS para indicar que está listo para recibir. Cuando la transmisión ha finalizado, el terminal puede bajar RTS. El modem debe ahora seguir procesando los datos que ha recibido y bajar CTS cuando esté listo para procesar el nuevo RTS. Cuando RTS es falsa, el terminal debe esperar a que CTS sea falsa antes de subir RTS para requerir una nueva transmisión. En la dirección contraria, en un enlace semi-duplex, el modem puede transmitir al terminal solo cuando RTS sea falsa.

En el otro protocolo utilizado por estas señales, cada dispositivo usa su salida independientemente para indicarle al otro cuando es posible enviar datos. CTS sigue con la misma función, pero RTS se redefine como el "listo para recibir" del DTE. Cada dispositivo comprueba la señal del contrario antes de transmitir.

RI es verdadero cuando aparece una señal de llamada en el canal de comunicaciones. La señal es verdadera cuando suena la llamada y falsa durante las pausas.

La última señal de control es CD. El modem sube CD cuando detecta una señal de la frecuencia esperada en la línea telefónica, indicando que se ha establecido una comunicación con un modem remoto.

2.1.2.5. Tensiones

Los niveles lógicos de RS-232 se indican con tensiones positivas y negativas en lugar de usar tensiones solo positivas como sucede con los 5 voltios de TTL y lógica CMOS o con los 3.3V de la tecnología de bajo consumo. A la salida de datos (TD) de una línea RS-232, un valor lógico 0 se define como igual o mas positivo que +5V, y un valor lógico 1 se define como igual o mas negativo que –5V. En otras palabras, se utiliza lógica negativa, dado que el valor mas negativo corresponde al 1.

Las líneas de control utilizan las mismas tensiones, pero con lógica negativa. Una tensión positiva indica que la función está On, y valores negativos indican que la tensión está Off.

Los chips de interfaz RS-232 invierten las señales. En el pin de salida de una UART, la señal para un bit 1 de datos o un control Off está cercano a 5V, lo cual resulta en una tensión negativa en el interfaz RS-232. Un bit 0 de datos o un control On tiene un valor cercano a 0V, resultando una tensión positiva en el interfaz RS-232.

Dado que el receptor de la señal RS-232 puede encontrarse al final de un largo cable, cuando la señal llega al receptor, su tensión puede haberse atenuado y tener ruidos

añadidos. Para permitir esto, las tensiones mínimas requeridos en el receptor son menores que en la transmisión. Cualquier entrada por encima de +3V se considera un 0 lógico de datos o un control On, y lo contrario para las señales por debajo de –3V. De acuerdo con el estándar, una entrada entre –3V y +3V queda indefinida.

El margen de ruido resultante de esto, es mayor que el que se obtiene con lógica TTL de 5V. Y muchos dispositivos tienen salidas RS-232 mucho mayores que ± 5V, como por ejemplo ± 9 o ± 12. Esto se transforma en márgenes de ruido muy anchos. La máxima variación permitida es de ± 15V, pero lo receptores deben ser capaces de soportar hasta ± 25V sin dañarse.

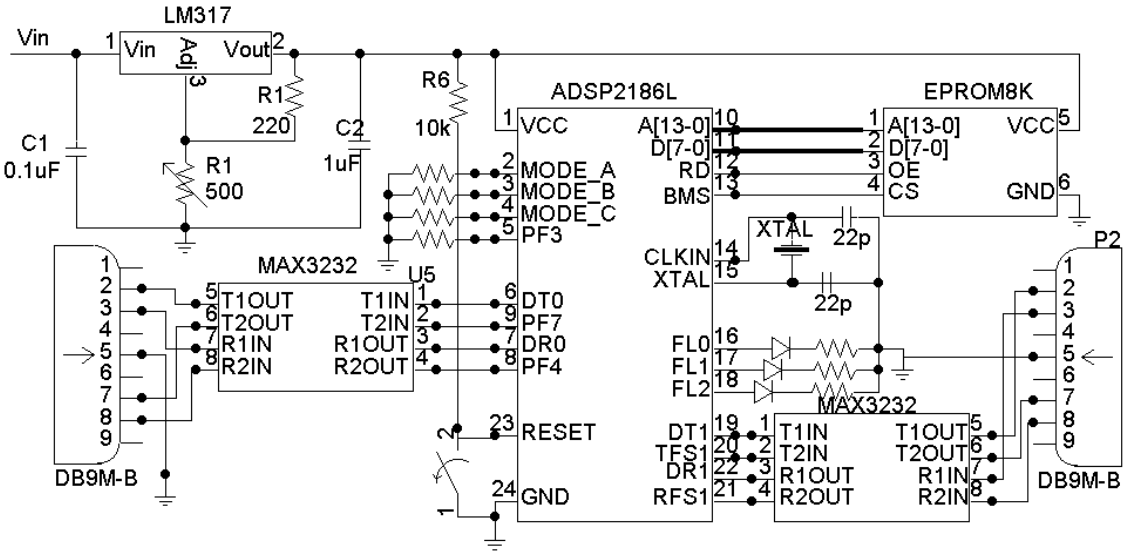
Otros dos términos que se usan en relación al estandar RS-232 son Marca y Espacio. Espacio representa un 0 lógico, y Marca es el 1 lógico. Hacen referencia a las marcas físicas y espacios hechos por los grabadores mecánicos que se usaban muchos años antes para almacenar datos binarios.

2.2. DESCRIPCIÓN DEL PROYECTO HARDWARE

El diseño del proyecto hardware consiste básicamente en una placa que dispone de un procesador DSP con memoria interna, que carga un programa externo desde una EPROM y se comunica con el PC mediante un puerto serie y con una placa idéntica a ella mediante otro puerto serie.

Para adaptar los niveles de tensión del procesador a los niveles de tensión RS-232 requeridos para la comunicación serie, se requieren además chips de tipo MAX232 que realicen esta adaptación. El procesador debe disponer además de un reset, activable mediante un pulsador, y debe ser configurado adecuadamente (lo cual implica alimentar la entrada de reloj mediante un cristal y poner otras entradas a 0 o a 1 mediante resistencias) y alimentado, y dispondremos también de ciertos LEDs de propósito meramente informativo, que pueden indicar estados tales como circuito encendido, transmitiendo o recibiendo.

El esquema en general del circuito puede verse en la siguiente figura:

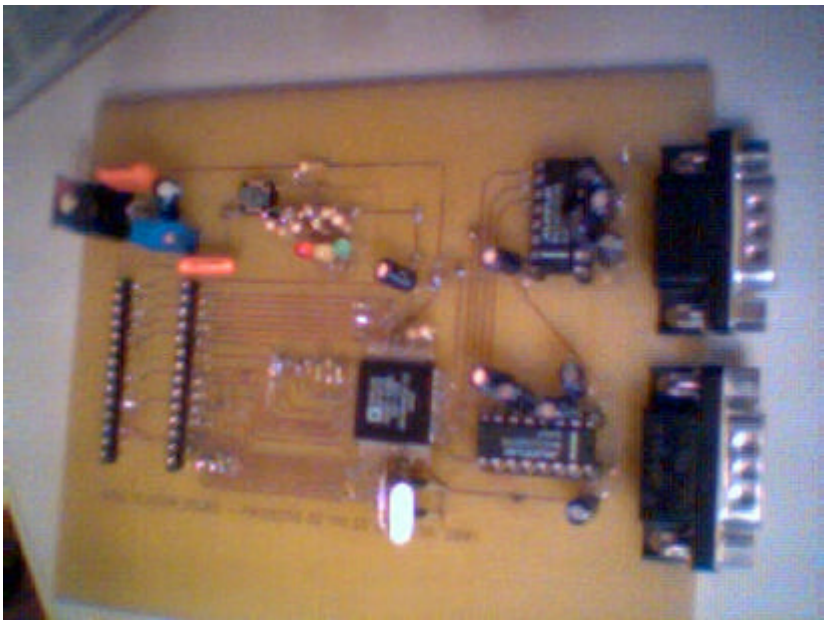


Este esquema está bastante simplificado. En realidad el chip ADSP-2186L tiene 100 patas, pero solo se han representado las mas importantes. Por otra parte, los chips MAX3232, tienen tanto alimentación como tierra, así como otros pines de configuración que se conectan con condensadores a tierra o alimentación. Tampoco

están representados los condensadores de desacoplo entre Vcc y GND de ninguno de los chips.

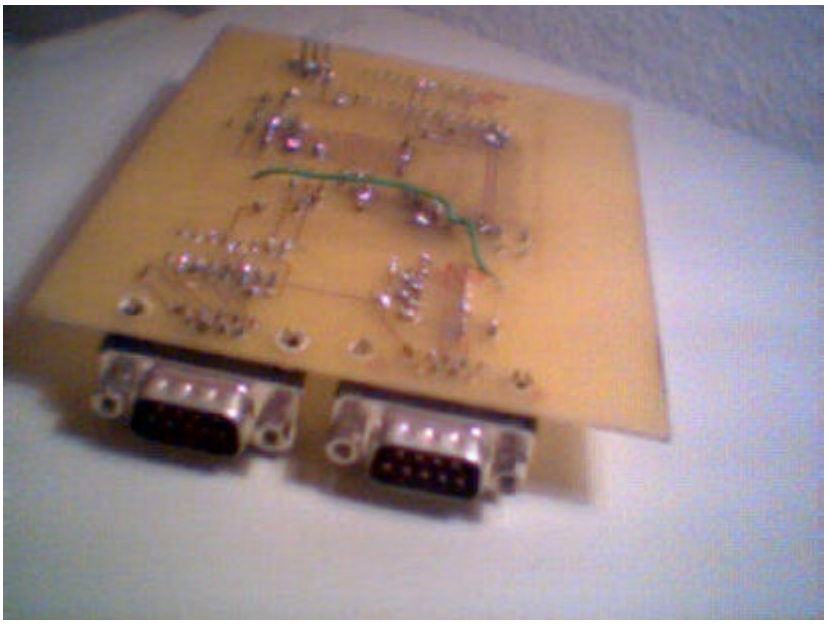
El diseño del rutado de la placa se realizó con el programa ACCEL Tango PCB, obteniendose el diagrama expuesto en la sección de planos como Plano 1.

El resultado del proceso de revelado de la placa y soldadura de componentes, dio lugar a la siguiente realización física de la placa:



Fotografía 2.2.1

Esta misma placa por detrás tiene el siguiente aspecto:



Fotografía 2.2.2

Se probaron todos los puntos de conexión unos con otros sin que se hallase el mas mínimo problema, y el pequeño programa de encendido de LEDs también funcionaba.

2.3. INTRODUCCIÓN A LA COMPRESIÓN DE DATOS

2.3.1. Que es la compresión de datos

La compresión de datos consiste en hacer un archivo o un flujo de datos de menor tamaño a base de predecir los bytes mas frecuentes y almacenarlos en menos espacio. Por lo tanto un compresor tiene al menos dos tareas diferentes: predecir las probabilidades de la entrada y generar códigos a partir de esas probabilidades, lo cual se consigue con un modelo y un codificador respectivamente. Opcionalmente, algunos datos como el sonido o las imágenes pueden ser transformadas o cuantizadas para lograr mayor compresión.

Un compresor puede ser con pérdidas o sin pérdidas. Con un compresor y un descompresor sin pérdidas, los archivos originales y los descomprimidos son idénticos bit a bit. Por otra parte, con otros tipos de datos, podemos obtener grandes beneficios en la eficiencia de la compresión desechando la mayor parte del material, pero sin que se pierda sin embargo demasiada calidad. Usamos compresión sin pérdidas para textos o datos binarios, como por ejemplo programas, y compresión con pérdidas para los datos como señales: audio, imágenes y video.

2.3.2. ¿Para que se usa la compresión de datos?

Para transmitir o almacenar la misma información en menos bits. Esto tiene muchas importantes y diferentes aplicaciones. Pero todas resultan ser la misma: evitamos el desperdicio de recursos. Recursos tales como espacio en disco, dinero o tiempo.

Desde el punto de vista del almacenamiento, la compresión se utiliza en copias de seguridad, para imágenes y sonido, datos internos de programas, o para comprimir el programa en si mismo. Pueden encontrarse aplicaciones reales en estandares de imágenes tales como el JPEG, o en estándares de audio como el MP3, los cuales han

ayudado mucho en la revolución multimedia. Otro aspecto muy importante son los archivadores, comúnmente llamados compresores, como ZIP o ARJ.

Pero, si en el almacenamiento la compresión de datos nos provee una ayuda importante, en la transmisión su importancia es aún mayor, dado que mientras mayor sea la compresión de los datos que se transmiten, mas se reduce el tiempo de transmisión y mas se abaratan por tanto las comunicaciones. A nadie se le ocurriría bajarse una imagen sin comprimir pudiendose bajar un .JPG que se ve prácticamente igual y va a ocupar 15 veces menos.

2.3.3. ¿Cómo funciona la compresión?

La compresión se basa en el hecho de que los datos son redundantes, que hasta cierto punto han sido generados siguiendo ciertas reglas, y que si podemos aprender esas reglas, podemos predecir los datos con una cierta exactitud. Claramente la compresión tiene 2 partes: una que averigua cuales son los símbolos mas frecuentes y otra que saca a la salida las “decisiones” de la primera parte.

Usando la terminología correcta diríamos que la primera parte se llama modelo y predice la probabilidad de la fuente, haciendo una distribución de probabilidades. Y la segunda parte es el codificador que hace y emite códigos basados en la probabilidades asignadas por el modelo. Mientras mas cercanas a la realidad sean las predicciones hechas por el modelo sobre los datos, mayor compresión lograremos alcanzar, por que asignaremos menos bits a la mayor parte de los datos.

Inmediatamente surgen las dos principales cuestiones de la compresión: ¿que reglas debemos seguir para asignar probabilidades a los símbolos? Y dada una distribución de probabilidades… ¿cuáles son los códigos mas pequeños que podemos asignar?. La teoría de la información es la ciencia que se encarga de responder a estas preguntas. Como veremos mas tarde, las respuestas son un modelo de Markov y la entropía respectivamente.

2.3.4. Texto y señales: compresión sin y con pérdidas

Hemos visto antes que podemos querer comprimir diferentes tipos de datos, tales como texto, bases de datos, programas binarios, sonido, imágenes y video. En la práctica distinguimos entre compresión de texto y compresión de señal. Hacemos esta separación por que las bases de datos y programas binarios tienen las mismas características que el texto. De la misma forma sonido, imagen y video son señales y por lo tanto comparten propiedades.

También hacemos esta separación por que para estos dos grupos utilizamos diferentes tipos de compresión. Esto viene de la naturaleza propia de los datos. Las señales digitales son representaciones imperfecta de señales analógicas, por lo tanto ya han sido comprimidas hasta cierto punto (pues hemos descartado algo de información al hacer la cuantización), y podemos descartar todavía mas información para lograr mayor compresión. Esto se logra con algoritmos de transformación y cuantización.

Digamos que tenemos un byte de una imagen, con el valor 65 que representa la cantidad de rojo de un pixel dado. Si al descomprimir ese byte se transformase en 66, no notaríamos la diferencia entre el rojo que había antes y un poquitín mas de rojo. Por otra parte, si en un texto el 65 (que corresponde al carácter ASCII ‘A’) se descomprimiese como 66 (que corresponde al carácter ASCII ‘B’) , el mensaje cambia. Debido a la naturaleza propia del texto, no podemos permitirnos errores.

Por lo tanto, usamos compresión sin pérdidas para texto, donde el archivo original debe ser exacto bit a bit al original, y compresión con pérdidas para las señales en las que algunos errores son aceptables y en muchos casos no llegan a detectarse. A pesar de todo, las señales también pueden comprimirse sin pérdidas, lo que sucede es que la compresión alcanzada será mucho peor que si se comprimen con pérdidas. En la mayor parte de los casos, la compresión de señales se basa en descartar tantos datos como sea posible pero tratando de retener la máxima calidad.

2.3.5. Clases de modelos

Existen diferentes clases de modelos, tales como modelos de estado finito, modelos gramaticales y modelos ergódicos, pero los mas utilizados sin duda alguna utilizan una variación de los modelos de Markov: modelos de contexto finito, que se basan en la siguiente propiedad: en una fuente de Markov de orden k, la probabilidad del símbolo actual depende solo de los k símbolos previos. Para conocer la probabilidad del símbolo actual tenemos en consideración lo que ocurrió antes, esperando que vuelva a suceder lo mismo.

Por ejemplo, en un modelo de contexto finito de orden 1 tomamos en consideración los símbolos que han aparecido después del contexto de actual de orden 1. Si tuvieramos que comprimir una ‘h’ precedida por una ‘t’ (contexto de orden 1), usaríamos las probabilidades pasadas solo cuando la ‘t’ estaba precediendo. En un texto en inglés, la ‘h’ es muy dada a aparecer tras una ‘t’ (there, that, the…).

Hay cuatro clases de modelos dependiendo de la regularidad con la que lo actualizamos: estático, semiestático, adaptativo y el inusual cuasiestático. El primero (estático) nunca actualiza las probabilidades de lo símbolos. Un modelo semiestático hace una primera pasada recogiendo todas las probabilidades, y una segunda para codificar todos los símbolos según las probabilidades de la primera pasada. El modelo adaptativo actualiza las probabilidades cada vez que se codifica un byte. Por último, el modelo cuasi estático actualiza las probabilidades cada K bytes, que son ajustados para acelerar el proceso a la vez que apenas afecta a la compresión lograda.

2.3.6. Calculando probabilidades con un modelo de contexto finito

Hemos dicho que usamos las probabilidades de un contexto, pero ¿cómo se hace esto?. Asumimos que la probabilidad de un símbolo es proporcional al número de veces que ha aparecido. En otras palabras, si todavía no ha aparecido, probablemente no lo hará.

Digamos que tenemos un modelo de contexto finito de orden 0 para este ejemplo: “bacb”. Orden 0 significa que no tomamos ningún contexto en consideración. La

frecuencia, que es el número de veces que un símbolo ha aparecido, es: ‘a’ > 1 , ‘b’ > 2, ‘c’ > 1. La frecuencia total es la suma de las frecuencias > 4. Por lo tanto la probabilidad de para cada símbolo es a > ¼, b > 2/4, c > ¼. Esto es una distribución de Probabilidades.

En el ejemplo anterior, si ahora apareciese un símbolo ‘d’, no podríamos codificarlo, por lo que no seríamos capaces de comprimir el archivo sin pérdidas. Por lo tanto, si queremos compresión sin pérdidas, el modelo debe asignar una probabilidad distinta de cero a todos los símbolos.

2.3.7. ¿Por qué usamos un modelo de contexto finito?

El objetivo de la compresión es predecir con exactitud los símbolos que aparecen mas a menudo para asignarles menos bits. Obviamente, si todos los símbolos tienen la misma probabilidad, no hay forma de comprimirlos. Si usamos un modelo de orden 0 (sin contexto) para conocer las probabilidades para este párrafo de texto, nos daríamos cuenta de que la mayor parte de los símbolos, en este caso letras y puntuaciones, no difieren demasiado. Obviamente, esto no nos va a permitir comprimir demasiado

El texto es sensible al contexto, sus probabilidades parecen generadas con un modelo de contexto finito. Por eso usando un orden mayor, las probabilidades se diferencian mas, dandole mayor importancia a unos pocos símbolos. En un modelo de orden 0, las probabilidades son mas o menos planas, pero si subimos a orden 3 las cosas mejoran mucho, por que por ejemplo nos damos cuentas que tras “tex” siempre aparece otra ‘t’ . Eso hace que las probabilidades de ‘t’ bajo este contexto sean muy altas, por lo que puede asignarse un código pequeño y lograr una compresión alta.

2.3.8. Entropía y las probabilidades que deben tener nuestros códigos

Codificar es hacer una serie de palabras de código a partir de una distribución de probabilidades. Asignamos una palabra de código a cada símbolo.

Por ejemplo, en el ejemplo anterior:

Símbolo	Probabilidad	Palabra clave binaria
A	¼	10
B	½	1
C	¼	01

Para la compresión de datos, las palabras clave deben tener ciertas propiedades. La primera es que deben ser códigos prefijos: ningún código puede ser el prefijo de otro código. Por ejemplo, el ejemplo anterior no cumple con esta propiedad, ya que 1 es el prefijo de 10, por lo que si tenemos 101, no sabemos si esto significa ‘bc’ o bien ‘ab’. Para que el código sea prefijo, podemos sustituir por ejemplo la palabra clave binaria correspondiente a la A por 00. También deben ser instantáneamente decodificables: debemos ser capaces de decodificar el símbolo en cuanto tengamos el código.

Según la teoría de la información, la mejor longitud media del código que podemos escoger para una probabilidad P es la entropía: -log2(P). Claramente, si tenemos un símbolo con una probabilidad de ½, el código debe tener la longitud de un solo bit, ya que -log2(1/2)=1. La probabilidad P la escoge el modelo. Si queremos que nuestros códigos sean instantáneamente decodificables y tengan la propiedad prefijo la mínima longitud del código es la entropía.

Podemos sentirnos tentados a pensar que podemos hacerlo mejor que la entropía, pero el teorema de Shannon sobre la codificación de fuentes sin ruido prueba que no podemos, al igual que lo prueban las desigualdades de Kraft.

El código ASCII es un ejemplo simple del paradigma de modelo y codificador. Se asigna una probabilidad plana (la misma probabilidad a cada símbolo) y también se alcanza la entropía. Por ejemplo, el byte ‘a’ (como cualquier otro) tiene una

probabilidad de 1/256, por lo que su probabilidad es –log2(1/256) = 8 bits, que es exactamente la longitud de código que usa ASCII.

2.3.9. Algunos codificadores

Los 2 codificadores de entropía principales son variaciones de Huffman y variaciones de codificación aritmética. Huffman se parece de alguna forma a los ejemplos mostrados antes. Hace códigos con longitudes variables, pero la longitud tiene que ser entera. Como en el ejemplo, los códigos tienen diferentes longitudes, pero ningún código tiene un número fraccionario de bits. Esto es un problema, ya que si por ejemplo la entropía dice que la longitud ideal del código es 1.5, el método de Huffman tiene que introducir redundancia en el código.

Por otra parte, la codificación aritmética es casi óptima si se compara con la entropía. En lugar de hacer un código para cada símbolo, se hace un único código para toda la entrada. Esto es lento, pero óptimo. Esto quiere decir que si la entropía de ideal de un código para cierto símbolo son 2.32 bits, el codificador aritmético lo codifica en 2.32 bits.

Al final todos los codificadores sacan bits a la salida. Así que tenemos rutinas para sacar bits directamente, pero estas rutinas también pueden emplearse directamente, por ejemplo si tenemos solo 16 símbolos posibles en un archivo que vamos a comprimir, podemos asignarles una probabilidad plana y codificar cada uno de ellos en 4 bits.

La salida simple de bits se usa cuando la velocidad importa. Si queremos mas compresión, conviene pasar a Huffman, que ofrece una buena relación compresión/velocidad. Para obtener la mayor compresión, usamos un codificador aritmético, que es mucho mas lento que Huffman (dado que pasa la mayor parte del tiempo actualizando el modelo). El codificador de rango es una variación del codificador aritmético, que es el doble de rápido. Según el tipo de modelo que vayamos a utilizar, conviene (aunque no es necesario) utilizar un codificador que se ajuste a las características del modelo, por que no tiene mucho sentido por ejemplo, usar un

codificador aritmético si estamos usando un modelo de tipo LZ, que logra una menor compresión a cambio de una mayor velocidad de ejecución.

2.3.10. Partes de un compresor

El modelo y el codificador son comunes a todos los compresores. Pero podemos utilizar otros algoritmos en nuestros compresores. Son los algoritmos de transformación y los algoritmos de cuantización. Si el compresor tiene algún paso con pérdidas, este está en la transformación o en la cuantización.

Los algoritmos de transformación se hacen para mejorar la estimación de probabilidades. Se usan muy a menudo en la compresión con pérdida de señales, de hecho son la columna vertebral de dichos métodos de compresión. FFT, DCT y wavelets, son algoritmos de transformación usados en la compresión de señal. Pero el texto también puede ser transformado, por ejemplo sustituyendo las letras mayúsculas por una bandera, reordenando el alfabeto, etc…

La cuantización consiste en reducir la precisión de los datos para almacenarlos en menos bits. Por ejemplo, para el sonido se almacena la presión del aire, y si usamos 4 bits en lugar de 8 para almacenar las muestras, estamos reduciendo el número de símbolos que representan la señal, y se puede seguir escuchando el sonido, aunque tal vez no con tanta nitidez.

2.3.11. Compresión de texto en el mundo real

Desafortunadamente, no ha habido ninguna discusión sobre una clasificación, pero se puede decir que en la práctica existen tres tipos diferentes de compresores: modelos complejos, variantes BWT y variantes LZ.

Los modelos complejos consiguen el mayor grado de compresión, pero son los mas lentos, y los que mayor número de recursos (memoria y CPU) utilizan. Los modelos complejos hacen una predicción lo mas certera posible de la entrada. Existen varias

clases de algoritmos, aunque los mas famosos son dos: DMC (Dynamic Markov Coding) comienza con un modelo de Markov de orden 0 que gradualmente se va extendiendo a medida que la compresión progresa, y PPM (Prediction by Partial Matching) busca una coincidencia del texto a ser comprimido en un contexto de orden n, si no se encuentra pasa a orden n-1, así hasta que encuentra la coincidencia o alcanza el orden 0. Estas técnicas, usadas junto a un codificador aritmético logran los mayores grados de compresión. Generalmente PPM obtiene una mayor compresión y es por lo tanto la técnica a la vanguardia en el estado actual de investigación sobre compresión (ppmd, ppmonstr…).

Las variaciones sobre BWT (Burrows-Wheeler Transform) comprimen menos que los modelos complejos pero por lo general son mucho mas rápidos. La Transformada de Burrows-Wheeler es una transformada sin pérdidas cuya salida contiene muchas cadenas del mismo símbolo. Esta redundancia es aprovechada para comprimir mediante otros métodos que no operan tan bien sobre una entrada normal, tales como MTF (Move To Front) junto con RLE (Run Lenght Encoding), seguido de un codificador. Lo que diferencia a la mayor parte de las implementaciones es la rutina de ordenación utilizada en la transformada. Estas técnicas están ganando popularidad últimamente debido a su buena relación de compresión / velocidad, por lo que cada vez son mas los compresores que hacen uso de ellas. Por ejemplo, bzip2 se basa en esta técnica y está sustituyendo progresivamente a gzip en el mundo Unix/Linux.

Los algoritmos derivados de LZ (siglas correspondientes a Lempel y Ziv, sus creadores), son las que menor grado de compresión logran, pero también son los mas rápidos. Lo que hacen es buscar coincidencias de los siguientes bytes a ser comprimidos entre los bytes anteriormente procesados, y si se hallan dichas coincidencias se codifican como parejas longitud-distancia. De no hallarse, se saca el byte procesado sin comprimir. Esto sucede así en las variantes de tipo LZ77, también existen las de tipo LZ78 que van construyendo un diccionario en donde se albergan todos los símbolos posibles y también las combinaciones entre ellos, por lo que la salida son siempre referencias a ese diccionario. Últimamente, también se han realizado avances en estas técnicas (como por ejemplo LZP de Charles Bloom), que siguen gozando de una gran popularidad y son las que se usan en la gran mayoría de los compresores que se usan cada día (ZIP, ARJ, RAR, ACE…). Por último, se puede mencionar también la técnica

RLE, que es la mas sencilla que se puede imaginar, tan solo consiste en codificar las series de símbolos repetidos, asignandoles un código que indica cuantas veces se repite ese símbolo. Esta técnica tan solo tiene interés en gráficos de pocos colores, en los que pueden aparecer hileras de pixeles del mismo color, pero para la compresión sin pérdidas de archivos en general no tiene mucho sentido. Por otra parte, esta técnica puede englobarse dentro de las técnicas de tipo LZ, ya que en el fondo es como si fuera una compresión LZ en la cual la distancia está siempre fija a 1.

En la práctica podemos combinar diferentes modelos y codificadores entre si, por ejemplo, podemos usar PPMC para la salida de LZP, o usar RLE antes de LZ77 para que las distancias halladas sean mas pequeñas.

2.4. EL MÉTODO DE COMPRESIÓN LZ

2.4.1. Teoría

En 1977 Abraham Lempel y Jacob Ziv presentaron su esquema basado en diccionario para la compresión de texto. Hasta la fecha, todos los algoritmos de compresión desarrollados eran principalmente compresores estáticos. El nuevo esquema fue llamado LZ77 (por razones obvias). Siempre sacaba a la salida desplazamientos y longitudes al texto previamente visto. También sacaba el siguiente byte después de la coincidencia, por que el contexto (los últimos bytes vistos) de este byte son los bytes que han coincidido, y si no es parte de la coincidencia (también llamada “frase”, o “match”), entonces no ha podido comprimirse, así que ¿para que perder tiempo (y espacio) intentando encontrar una coincidencia para el?

En 1982 James Storer y Thomas Szymanski, basandose en el trabajo de Lempel y Ziv, presentaron su esquema, LZSS. La diferencia principal es que a la salida, LZ77 siempre sacaba un par distancia/longitud, incluso cuando la coincidencia era de un solo byte (y en este caso estabamos usando mas de 8 bits para representar un byte, por lo que se perdía compresión), así que LZSS emplea otro truco para mejorarlo, usa flags de bit, que son un simple bit que cuenta si lo que viene es un byte sin comprimir (llamados literales o crudos “raw”), o bien un par desplazamiento/longitud. Esto es lo que se usa hoy en día, pero LZSS se conoce comúnmente como LZ77, por lo que lo llamaremos así de ahora en adelante.

La teoría que hay detrás de esta técnica es simple e intuitiva. Cuando se encuentra una coincidencia (o match, o frase, es decir, un grupo de bytes que ya han aparecido antes en la entrada), en lugar de escribir esos bytes se escribe el desplazamiento y la longitud de la repetición: aquí comienza y ocupa esto.

Es un esquema basado en diccionario, por que aunque no se construye un diccionario como en otros métodos basados en LZ78 (un nuevo método propuesto por Lempel y Ziv un año mas tarde que el original), se usa como diccionario la misma entrada (llamada “ventana deslizante” por que tiene un tamaño máximo y al sobrepasarlo vamos

perdiendo de vista los primeros bytes, como si solo pudiesemos ver a través de una ventana que se va desplazando por la entrada) y se hacen referencias a esa misma entrada (mediante las parejas desplazamiento/longitud).

2.4.2. Ejemplo

Imaginemos que estamos comprimiendo el siguiente texto: “ab ab”. Leemos “ab “ y lo escribimos sin comprimir, entonces leemos “ab” y escribimos: “en la posición 0 hay 2 bytes repetidos”. ¿Cómo funcionaría la compresión? Es fácil, primero leemos “ab “, y entonces la pareja desplazamiento y longitud, y copiamos los bytes desde ahí.

Paso 1: Obtenemos “a” -> “a”
Paso 2: Obtenemos “b” -> “ab”
Paso 3: Obtenemos “ “ -> “ab “
Paso 4: Obtenemos la pareja posición 0 y longitud 2. Copiamos 2 bytes desde la posición 0 de la salida (que es ahora nuestro diccionario): > “ab ab”

¿Pero como puede saber el descompresor si hay un par desplazamiento/longitud o un byte sin comprimir?. Simplemente se usa un prefijo, que es un bit que conmuta entre los dos casos y nos indica que tipo de datos le siguen. Por ejemplo, si es un 0 tenemos un byte sin comprimir, y si es un 1 hay un par desplazamiento/longitud. Estos prefijos también se conocen como banderas (flags).

El par desplazamiento y longitud se llama palabra código (codeword), o par código (codepair). Es un grupo de bits que contiene cierta clase de información que es usada por ambos el compresor y el descompresor.

Por lo tanto las salidas de LZ77 son de tres clases:

1 – Literales: Tan solo son bytes sin comprimir
2 – Palabras códigos: En nuestro caso son parejas de desplazamiento y longitud
3 – Flags: Nos cuentan si lo que sigue es un literal o una palabra código.

Ahora veamos de nuevo el ejemplo con la salida que se produce:

Paso 1: Obtenemos “a”. No hay match. > Flag 0 + Literal “a”
Paso 2: Obtenemos “b”. No hay match. > Flag 0 + Literal “b”
Paso 3: Obtenemos “ “. No hay match. > Flag 0 + Literal “ ”
Paso 4: Obtenemos “a”. Hay match. > Flag 1, código: desplazamiento=0, longitud=2

Podemos observar que las banderas solo tienen dos estados: 1 o 0. Así que solo necesitamos un bit para representarlas. No podemos (es decir, no debemos) sacar los flags como un byte completo cada uno, así que tenemos que trabajar con bits. La salida de esta compresión se llama flujo de bits, por que es un flujo de símbolos de longitud variable, y la unidad mínima es el bit.

2.4.3. Ventana deslizante

Si volvemos a mirar el ejemplo, podemos preguntarnos: ¿hasta donde tenemos que buscar las coincidencias?. Vamos mirando hacia atrás, los datos que ya hemos procesado. Esto se llama ventana deslizante. Es un buffer que almacena los bytes que están antes que la posición actual del archivo (o flujo de datos) que estamos comprimiendo. Cada byte que sacamos como literal es añadido a la ventana deslizante, así como todos los que forman una coincidencia.

Veamos de nuevo el ejemplo:

Paso 1: Obtenemos ‘a’, Ventana “” (vacía). No hay match > Flag = 0, Literal ‘a’
Paso 2: Obtenemos ‘b’, Ventana “a”. No hay match > Flag = 0, Literal ‘b’
Paso 3: Obtenemos ‘ ’, Ventana “ab”. No hay match > Flag = 0, Literal ‘ ’
Paso 4: Obtenemos ‘a’, Ventana “ab “. Hay match > Flag=1, código: desplazamiento=0, longitud=2

Como se puede observar, al buscar coincidencias (matches), comparamos los datos que tenemos en nuestra ventana con los datos que hay en la posición actual.

Dado que el buffer de entrada y la ventana deslizante van a compartir los mismos datos, (del archivo o flujo que va a ser comprimido), podemos usar este buffer de entrada como ventana deslizante. Al hacer esto, tan solo tenemos que preocuparnos por el puntero a la posición actual, y la ventana deslizante está desde el comienzo del buffer hasta justo antes de dicho puntero.

Ahora hablemos sobre la ventana deslizante. ¿Qué tamaño tiene? De hecho podríamos trabajar con el archivo completo que vamos a comprimir, pero hemos de pensar que tenemos que especificar la posición del match mediante el desplazamiento. El desplazamiento que se usa no es desde el comienzo del buffer hasta llegar a la posición de la coincidencia, sino desde la posición actual hacia atrás (por eso se le llama a este desplazamiento “distancia”). Así que en el ejemplo el desplazamiento es 3 en lugar de 0, por lo que al descomprimir, el descompresor obtiene una distancia 3, se la resta al valor actual del puntero de la posición actual y de esta forma obtiene la posición exacta a partir de la cual copia los bytes especificados por la longitud.

Como se puede suponer, mientras mayor sea la ventana deslizante mas bits necesitaremos para salvar la distancia, así que tenemos que escoger una longitud apropiada para nuestra ventana.

2.4.4. Longitudes

Otra cosa que debemos elegir es la longitud de la longitud, es decir, cuantos bits van a usarse para especificar la longitud de la coincidencia, y por lo tanto cuantos bytes previos se pueden copiar como máximo con una solo match. Al variar los bits para la longitud y la distancia, podemos mejorar la compresión en ciertos archivos y empeorarla en otros, así que si queremos comprimir un solo archivo debemos probar varias combinaciones para quedarnos con el mejor valor, pero para un compresor general debemos alcanzar un equilibrio.

La longitud mínima de coincidencia es otro asunto importante. Si escogemos por ejemplo 13 bits para la distancia (ventana deslizante de 8K) y 5 bits para la distancia (longitudes entre 0-31), necesitaremos 18 bits para expresar el codepair, por lo que no

tiene sentido buscar un match de 2 bytes, así que decidimos que el mínimo match debe ser de 3 bytes. Dado que la longitud que puede expresarse con 5 bits va de 0 a 31, sumamos esa longitud mínima y podemos expresar longitudes de 3 a 34. Para codificar eso, restamos 3 a la longitud hallada al comprimir, y al descomprimir se suma 3 una vez leída dicha longitud en el codepair.

2.4.5. Marcador de fin

Ahora ya sabemos como funciona la descompresión, pero el descompresor debe saber cuando debe parar. Esto puede hacerse de dos formas:

1 – Mediante un símbolo especial que marque el final de los datos.
2 – Indicando la longitud de los datos antes de mandar el flujo de bits comprimidos.

Para usar el primer método, podemos reservar una determinada longitud o distancia (o combinación de ambos) de tal forma que al acabar la compresión se emite una última palabra código que contiene esa longitud, y marca el final del archivo. El descompresor, al leer cada codepair comprobará si la longitud coincide con esa reservada, y de ser así, dejará de comprimir en lugar de copiar bytes como con los demás codepairs.

2.4.6. Trabajando con bits

Dado que tenemos que tratar símbolos de longitud variable, el trabajo con bits es inevitable, pero el procesador está diseñado para trabajar con bytes completos, por lo que la manipulación bit a bit es mas lenta. Tenemos por lo tanto otra elección que hacer y es decidir si vamos a trabajar exclusivamente con bits, exclusivamente con bytes o una mezcla de ambos.

Trabajar con bits es lo mas sencillo, tan solo tenemos que preocuparnos de ir colocando bit a bit y cada 8 bits sacar un byte a la salida, así como justificar los últimos bits hacia la izquierda, pero esto es poco eficiente, tanto para el compresor como para el descompresor.

Para trabajar con bytes, tenemos que asegurarnos que los codepairs ocupan un número entero de bits, por lo que la suma de los bits de la distancia y la longitud debe ser un múltiplo de 8. Una posibilidad muy empleada puede ser emplear 12 bits para la distancia (ventana de 4K) y 4 para la longitud (3-18), entonces tan solo nos quedan las banderas que ocupan 1 bit cada una, por lo que cada 8 banderas con sus correspondientes bytes literales o codepairs, se juntan en el mismo byte, tras el cual vienen los 8 elementos dependiendo de esas banderas. De esta forma leemos los literales como bytes, y los codepair como words, y para separar la longitud y distancia tenemos que efectuar una operación and y un desplazamiento, mientras que para los bits de banderas se efectúa un solo desplazamiento para obtener cada uno de ellos. Esta forma de trabajar es mas eficiente, pero limita mucho la codificación de los codepairs, lo cual puede redundar negativamente en la compresión.

La tercera alternativa es una mezcla de las dos anteriores. En este caso diferenciamos los literales de las banderas y codepairs. Los literales van por su lado y las banderas y codepairs van juntos en el flujo de bits. Se lee el flujo de bits byte a byte, y se va decodificando según sea necesario. Si encontramos un bit correspondiente a un literal, entonces se lee el siguiente byte y se toma como literal, si por el contrario se encuentra un codepair, se siguen leyendo bits del flujo de bits hasta que se nos acabe el byte, momento en el que tomamos el siguiente byte como perteneciente al flujo de bits para seguir decodificando el codepair. Los bits correspondientes al flujo de bits que no son literales se conocen con el nombre de tag-bits (etiquetas). Esta alternativa es a la vez bastante eficiente y deja libertad total para codificar los codepairs, lo único malo es que es algo mas complejo de programar.

2.4.7. Posibles mejoras

Ya hemos visto el funcionamiento básico del algoritmo, pero se pueden hacer varias cosas para mejorar la eficiencia de compresión y la velocidad.

Para mejorar la velocidad, el principal problema es la búsqueda de coincidencias, para la cual lo único que hemos considerado hasta este momento ha sido la búsqueda de fuerza bruta (ir comparando byte a byte hacia detrás con el byte actual que vamos a tratar de comprimir, y por cada uno que encontremos, comparar hacia delante incrementando mientras llevamos la cuenta de cuantos bytes coinciden. Existen diversas técnicas para aumentar la velocidad de la búsqueda. Lo mas usual consiste en construir un árbol sufijo (suffix tree) o un árbol binario a la vez que se va procesando la entrada, pues estos algoritmos garantizan la búsqueda de la una determinada cadena de longitud M realizando M comparaciones, en lugar de realizar M*N (siendo N el tamaño total de la entrada, en nuestro caso el tamaño de la ventana deslizante), a cambio de ir construyendo el árbol, lo cual es una operación que también consume algo de tiempo (aunque mucho menos que el que se gana). Lo malo de esta técnica es que necesita memoria para almacenar el árbol, por lo que tenemos un intercambio de velocidad a cambio de memoria.

Para mejorar la compresión, hay dos detalles podemos tener en cuenta. Uno de ellos es la codificación de la longitud y la distancia en los codepairs. Hasta ahora hemos considerado una longitud fija de estos campos, pero es posible emplear otras codificaciones de longitud variable que se adapten mejor a nuestras necesidades. Por ejemplo, se puede emplear un codificación gamma, consistente en que tras cada bit, hay otro que indica si existen mas bits (por ejemplo con un 1) o el anterior era el último (0). Así, una distancia de 5 (101) quedaría codificada como: 110110, mientras que una distancia de 142 (10010010) se codifica como 1101011101011100, en lugar de usar 13 bits si el tamaño de nuestra ventana es de 8 Kbytes. Este método en particular incrementa mucho el número de bits necesarios si la cifra a codificar es grande, pero si es pequeña la codificación se realiza en menos bits y por lo tanto ganamos en compresión, por lo que tan solo se debe emplear esto si estamos seguros de que estadísticamente van a ser mucho mas frecuentes las distancias cortas que las largas. Esto último no es fácil que se cumpla con las distancias, pero si con las longitudes, ya

que son mucho mas comunes las coincidencias de pocos bytes que las de muchos. También es posible combinar esta técnica con un número fijo de bits, de forma que por ejemplo se emplean siempre 6 bits y el resto de la distancia se codifica con gamma para no castigar tan duramente las distancias largas. Otras posibles combinaciones son emplear distintos tipos de codepairs, uno para distancias cortas y otro para distancias largas, distinguiendolos con otras banderas extra, o bien emplear un número fijo de bits que indican cuantos bits mas forman parte de la distancia.

El segundo detalle que podemos tener en cuenta es que puede ser conveniente no usar la mejor coincidencia hallada, sino esperar para ver si podemos conseguir una todavía mejor mas adelante. Por ejemplo, si vamos a comprimir el texto: “curry urrent current”, cuando alcancemos la segunda “c”, encontraremos que la cadena coincidente mas larga es “curr”, de longitud 4, pero es mas conveniente sacar el carácter “c” como literal ya que de esta forma encontramos el match “urrent”, de longitud 7 justo después. A esto se le conoce como “Lazy coding”, e implica que debemos postergar la decisión de emitir un codepair a la salida antes de comprobar la longitud del siguiente codepair. Si en lugar de tener en cuenta solamente la coincidencia que puede hallarse en la siguiente posición, sino todas las coincidencias que pueden realizarse con cada uno de los caracteres que abarca el match recién hallado, estamos realizando lo que se conoce como “flexible parsing”, que proporciona resultados óptimos para esta técnica a cambio de aumentar la bastante la complejidad del código y el tiempo de búsqueda.

3. MEMORIA JUSTIFICATIVA

3.1. HARDWARE

3.1.1. Primer Paso: Elección del microprocesador a utilizar

Esta elección no es demasiado crucial en el sentido de que tenemos varios grados de libertad: elección de procesador, técnicas de compresión, velocidad de los datos a través del cable, etc..., pero conviene hacer una elección que facilite el diseño de las placas a la vez que mantenga una cierta versatilidad a la hora de escoger las técnicas de compresión y sin desperdiciar las características extras que incorpore el chip.

3.1.1.1. CARACTERÍSTICAS DESEADAS:

Buscamos un microprocesador o DSP con dos puertos serie programables independientemente y con canales DMA que permitan ejecutar un programa a la vez que se realizan las dos transferencias independientes de datos (comunicación entre la tarjeta compresora y descompresora, y comunicación con el PC). También necesitamos una mínima potencia de procesamiento para hacer posible la compresión en tiempo real manteniendo un flujo constante de datos en la comunicación entre placas, pero tampoco queremos que sobre demasiada potencia, por lo que se estima que un sistema de 16 bits es suficiente para nuestra aplicación, por lo que estos sistemas se preferirán a otros análogos de 32 bits. Otra característica deseable es la inclusión de memoria RAM dentro del chip, lo cual nos ahorrará componentes adicionales en la placa.

3.1.1.2. BÚSQUEDA DE INFORMACIÓN:

Se realizó una búsqueda en Internet de las características técnicas de microprocesadores de 6 de las principales marcas del negocio. No se incluyeron Intel y AMD debido a que están centradas sobre todo en la producción de microprocesadores

orientados a ordenadores, y al ser la programación de microprocesadores 80x86 ya conocida, los principales procesadores de estas marcas (y otros clónicos de Intel, tales como Cyrix) carecen de valor didáctico. A continuación se detallan las conclusiones de esa búsqueda para cada marca en concreto:

3.1.1.3. NEC:

Al buscar información sobre microprocesadores, obtenemos que tenemos a nuestra disposición varias familias: Vr4100, Vr4300, y Vr10000/12000.

De todas estas familias, la clase mas baja en cuanto a prestaciones es la Vr4100, pero a pesar de que los modelos Vr4100, Vr4101, Vr4102 y Vr4111 están ya obsoletos (ya no se producen, aunque podrían estar disponibles a través de terceros), los miembros de esta familia se basan en una arquitectura RISC de 32/64 bits, con instrucciones MISP, lo cual los hace demasiado potentes para la aplicación que se les va a dar.

A modo de ejemplo, el modelo Vr4181 incorpora, de acuerdo con el documento U14272EJ1V0UMJ1 como características propias convertidores A/D y D/A para entrada y salida de sonido, así como 2 canales DMA reservados también para el sonido, y tres puertos serie (el principal, el secundario y el temporizado), así como un interfaz de teclado que soporta una matriz de 8x8, y un interfaz para conectar con una pantalla tipo LCD, todos integrados dentro del chip. Y el modelo Vr4111 (cuyo modelo a 70 MHz está en proceso de liquidación, estando el modelo a 90MHz ya obsoleto), de acuerdo con el documento 13137e20 también tiene las interfaces de audio, teclado y además interfaz de panel táctil.

La siguiente serie Vr4300, ya se basa en procesadores de 64 bits, con lo cual quedan definitivamente descartados.

Resumiendo: Descartados por ser demasiado potentes y disponer de demasiada circuitería adicional que quedaría sin usarse.

3.1.1.4. HITACHI:

Los procesadores de Hitachi forman parte de una familia denominada "Super Hitachi". Todos los modelos de esta familia tienen una arquitectura de 32 bits que los convierte en fácilmente descartables para nuestro proyecto.

Los miembros de menor capacidad de esta familia son los modelos 7020/7021, pertenecientes a la subfamilia SH-1. En esta ocasión las características técnicas se ajustan bastante bien a lo que necesitamos, exceptuando algunos pequeños detalles, como el hecho de que existan 4 canales DMA, pero eso es
algo sin demasiada importancia. Mas importante es el hecho de que no contenga mas que 1Kb de memoria RAM en el chip. Los otros 2 miembros de la familia SH-1 (los modelos 7032/7034) son bastante parecidos a estos 2, pero incorporan también conversores A/D y D/A, por lo que no merece la pena que
los consideremos.

3.1.1.5. MOTOROLA:

Los microprocesadores de la serie 68x00 no tienen puertos serie incorporado (necesitan utilizar una DUART externa, por ejemplo), y tampoco tienen memoria incorporada en el chip. Además a partir del 68020 tienen una arquitectura de 32 bits. Por otra parte los PowerPC también tienen arquitecturas de 32 o 64 bits, son mas potentes que los anteriores (pueden llegar hasta los 300MHz en la serie MPC6XX o hasta los 500MHz en el caso de la serie MPC 7XXX) y al igual que los anteriores, están mas bien orientados a ordenadores, pero no nos sirven para nuestra aplicación.

Si consideramos los microcontroladores (que son el punto fuerte de la compañía) de 16 bits (series M68HC12 y M68HC16), vemos que estos si traen los puertos serie, pero todos ellos tienen también convertidores de señales analógicas (al menos 6 canales, aunque lo normal es 8), y tampoco resultan demasiado adecuados para lo que se pretende.

3.1.1.6. ZILOG:

Los microprocesadores de esta compañía son los descendientes del mítico Z80, todo un superventas en el pasado que obtuvo el control principal de 3 de las principales marcas de ordenadores domésticos de la pasada década, (Spectrum, Amstrad y MSX), tras lo cual siguió vivo muchas veces destinado como chip secundario de sonido tanto en máquinas recreativas arcade como en consolas de videojuegos como la Megadrive o la Neo-Geo.

El sucesor del Z80 es el Z80180, cuya versión mejorada, el Z8S180 resulta muy atractiva para el proyecto. Este chip incorpora dos puertos serie y dos canales DMA programables independientemente que permiten alcanzar velocidades de hasta 512Kbps, y no tiene características extra que vayan a ser desperdiciadas si se utiliza para el proyecto.

El problema de este chip es que sigue siendo, al igual que el Z80 (con el cual mantiene compatibilidad de código), un procesador de 8 bits cuyas instrucciones tardan 3 ciclos en ejecutarse (en lugar de un solo ciclo como en todos los demás procesadores mas avanzados, que tienen estructura pipeline), y aunque tiene un modelo que alcanza los 33 MHz, la velocidad podría ser demasiado baja para realizar la compresión en tiempo real, por lo que es mejor descartarlo y utilizar otro mas potente que aunque no se utilice al 100% de su capacidad puede dar mayor versatilidad a la hora de escoger el método de compresión y poder obtener mejores resultados. Otro problema es que este chip tampoco incorpora memoria dentro, por lo que habría que incluir chips externos en la placa.

Existe una versión mas avanzada del chip de Zilog, el Z380, pero es un híbrido entre un procesador de 8 bits compatible con el Z80 y otro nuevo de 32 bits, solo puede alcanzar los 18 Mhz, sigue necesitando 2 ciclos como mínimo para ejecutar cada instrucción y además es un descendiente del Z80180, por lo que no posee las características avanzadas del Z8S180, como los puertos serie y los canales DMA, y esto lo descalifica totalmente para el proyecto.

RESUMIENDO: Poco potentes

3.1.1.7. TEXAS INSTRUMENTS:

La compañía lider en el mercado del DSP dispone de una amplia gama posibilidades. Si descartamos los DSPs de coma flotante (casi todos los algoritmos de compresión trabajan con enteros), y nos quedamos solo con las gamas mas bajas, al examinar el documento SPRU018 observamos que existe un modelo de la gama TMS320C1X que incorpora 2 puertos serie, que es el modelo TMS320C17. Este modelo ofrece buenas características para el proyecto, aunque tiene el problema de que carece de RAM interna, y tal vez le sobran los puertos para coprocesador que también incorpora.

3.1.1.8. ANALOG DEVICES:

Los DSPs de la serie ADSP21xx incorpora una serie de características que los hacen muy indicados para el proyecto: tiene los dos puertos serie con sus canales DMA, son DSPs de 16 bits e incorporan memoria RAM dentro del chip, a la vez que carecen de características extra, tales como interfaces de audio, teclado o coprocesador, o convertidores analógicos. Los diferentes miembros de la familia se diferencian principalmente por la cantidad de memoria y la velocidad.

El procesador escogido en un principio fue el ADSP-2187L, que incorpora una memoria RAM de 160K, dividida en 96K para programa (32Kwords de 24 bits) y 64K para datos (32Kwords de 16 bits), pero por falta de disponibilidad al final el procesador adquirido y con el cual he trabajado toda la parte hardware es la versión ADSP-2186L, que tan solo tiene 8Kwords en cada una de las memorias, por lo que la cantidad de memoria disponible se divide por 4, lo cual nos obliga a optimizar mucho mas el aprovechamiento de la memoria y nos restringe las posibilidades en cuanto a técnicas de compresión.

3.1.2. CONSIDERACIONES DE DISEÑO

Una vez que hemos escogido el procesador, hay que analizar la información disponible sobre dicho procesador para ser capaces de diseñar un circuito eficiente y funcional.

3.1.2.1. Alimentación

El primer detalle que tenemos que tener en cuenta es que el ADSP-2186L es un chip de bajo consumo, por lo que debe alimentarse a 3.3V en lugar de utilizar los 5V habituales. Para obtener esta tensión, utilizamos un regulador LM317 con la configuración que aparece en la figura 3.1.2.1:

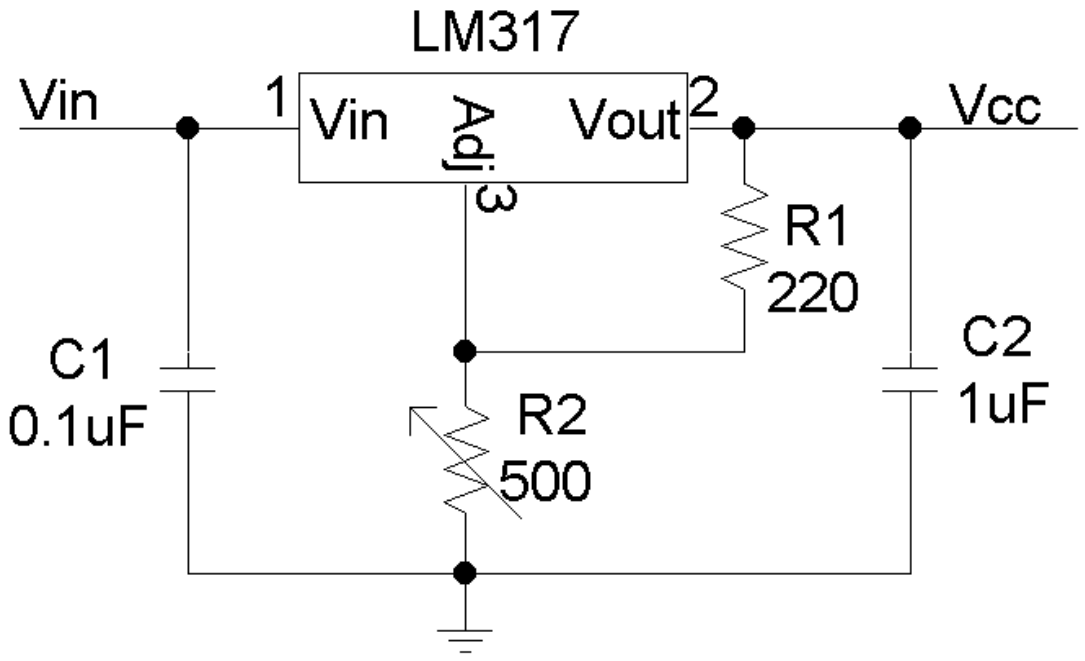


Figura 3.1.2.1: Esquema del circuito de alimentación

Mediante este diseño, la tensión a la salida se calcula como:

$$Vcc = 1.25\ V * \left(1 + \frac{R1}{R2}\right) + Iadj\ (R2)$$

siempre y cuando la entrada supere un cierto umbral. El término Iadj suele ser muy pequeño y se puede despreciar, por lo que para obtener a la salida Vcc = 3.3V, siendo R1 = 220 Ω, en el potenciómetro habrá que ajustar R2 = 360Ω. Es mejor emplear un potenciómetro para evitar problemas derivados de la tolerancia de las resistencias.

3.1.2.2. Señal de reloj

El ADSP-2186L puede ser temporizado por un cristal o por una señal de reloj con niveles compatibles TTL. Si se utiliza un reloj externo, debe conectarse la señal al pin de entrada del procesador CLKIN, dejando sin conectar el pin XTAL. La señal debe ir a la mitad de la velocidad del procesador, por lo que un reloj de 20MHz obtiene de rendimiento un ciclo de procesador de 25ns, lo cual equivale a 40MHz.

Al incluir el ADSP-2186L un oscilador dentro del mismo chip, se puede emplear también un cristal externo, conectado entre los pines XTAL y CLKIN, con dos capacidades puestas a tierra como se indica en la siguiente figura 3.1.2.1, también a la mitad de la velocidad del procesador.

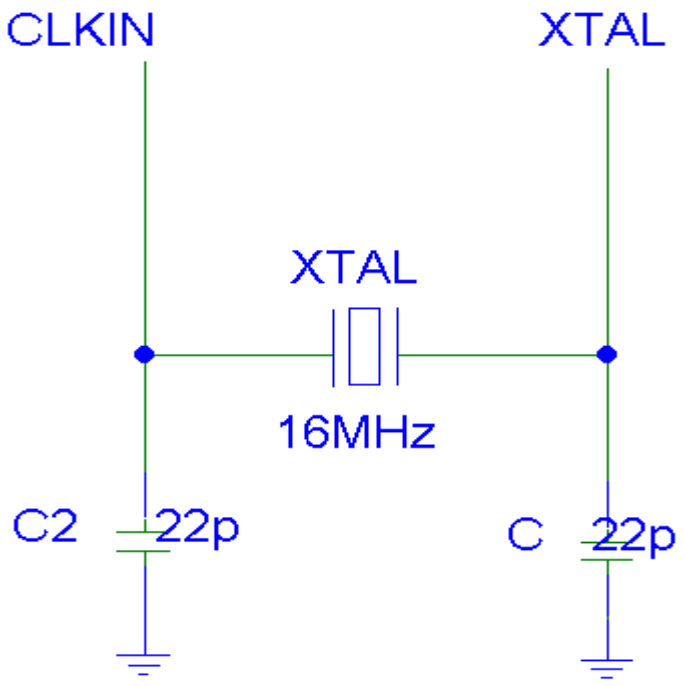


Figura 3.1.2.1: Entrada del reloj

La velocidad del escogida para el cristal no debe superar la mitad de la velocidad del microprocesador, y dado que el modelo adquirido fue el ADSP-2186LKST-133, con una velocidad de 33MHz, se optó por un cristal de 16 MHz que proporciona 32 MHz de velocidad para el micro (la siguiente alternativa era usar 16.9 MHz, lo cual se sale de nuestras posibilidades.

3.1.2.3. Reset

El reset del procesador se produce poniendo a nivel bajo la tensión de la señal RESET durante un periodo mínimo de 2000 ciclos de reloj. Si el reset se activa mediante un circuito externo RC, es conveniente usar un Schmidt trigger para provocar el reset, aunque la entrada ya incluye algo de histéresis. Dado que el reset que vamos a usar en el circuito es externo, esto no es necesario.

La parte del circuito correspondiente al reset consiste simplemente en una resistencia de 10KΩ que se conecta a Vcc en un extremo y a la señal reset en el otro, de forma que la señal está normalmente puesta a 1, a la vez que se une a tierra mediante un pulsador, por lo que el reset pasa a nivel bajo mientras se presiona el pulsador. 2000 ciclos de CLKIN a 32MHz equivalen a 0.0625ms, por lo que cualquier pulsación asegura un reset completo.

3.1.2.4. Arquitectura de memoria

El ADSP-2186L cuenta con una gran variedad de interfaces de memoria y periféricos. Los grupos funcionales clave son la memoria de programa (Program Memory), la memoria de datos (Data Memory), la memoria de byte (Byte Memory) y la E/S (I/O).

- Memoria de programa: Es un espacio de 24 bits de anchura que sirve para almacenar tanto código como datos. Disponemos de 8K words de RAM de memoria de programa dentro del chip, y la posibilidad de acceder hasta 2 bloques mas de 8K words de espacios de memoria externa usando el bus externo. Se pueden leer tanto una instrucción como un dato desde la memoria de programa interna en un solo ciclo.

- Memoria de datos: Es un espacio de 16 bits de anchura que se utiliza para el almacenamiento de variables y registros de control mapeados en memoria. El ADSP-2186L tiene 8K words de memoria de datos dentro del chip, consistentes en 8160 direcciones accesibles por el usuario y 32 registros mapeados en memoria. También se pueden soportar hasta 2 memorias externas de 8K words a través del bus de datos externo.

- Memoria de byte: Provee acceso a un espacio de memoria de 8 bits de ancho a través del puerto BDMA (Byte Direct Memory Access). El interfaz de memoria de byte permite acceder a 4Mbytes de memoria utilizando 8 líneas del bus de datos como líneas de dirección adicionales. Esto confiere al puerto BDMA un rango efectivo de 22 bits. Al arrancar, el DSP puede cargar automáticamente el código de arranque del sistema desde la memoria de byte, posibilitando así la carga de un programa externo mediante una EPROM.

- Espacio E/S: Permite acceder hasta 2048 localizaciones de 16 bits. Está pensado para ser usado en comunicaciones con dispositivos paralelos tales como convertidores de datos y registros externos.

El controlador DMA de memoria de byte permite la carga y almacenamiento de instrucciones de programa y datos utilizando la memoria del espacio de byte. El circuito BDMA es capaz de acceder a la memoria de byte mientras el procesador está operando de modo normal, y tan solo roba un ciclo de DSP por cada palabra transferida, ya sea de 8, 16 o 24 bits.

Esta organización de la memoria se da cuando tenemos el chip configurado en el modo Full Memory, en el cual el ADSP-2186L es el que controla la memoria y sus periféricos, pero existe otro modo llamado Host que permite la comunicación del DSP con un sistema anfitrión, con el cual se comunica mediante el puerto IDMA (Internal Memory DMA), teniendo acceso a la memoria interna tanto de programa como de datos, aunque no a los registros mapeados en memoria. Este modo no lo utilizamos en el proyecto.

De todos los tipos de memoria existentes, tan solo vamos a utilizar la memoria interna tanto de programa como de datos y la memoria de byte, que vamos a utilizar únicamente para almacenar el programa que se ejecuta al arrancar el procesador.

Por lo tanto, tenemos que conectar una EPROM con el DSP, mediante las líneas A13-A0 para el bus de datos y las líneas D15-D8 para el bus de datos (esto es así por que la EPROM es una memoria de 8 bits de anchura). También tenemos que conectar dos señales fundamentales para la EPROM, que son OE (que se conecta al pin RD del procesador) y CE (que se conecta a BMS).

3.1.2.5. Configuración y arranque

Existen varias formas de arrancar el procesador, que se controlan mediante los pines Mode A, Mode B y Mode C. Estos pines solo se evalúan en el arranque del procesador, quedando libres para ser usados como pines de entrada o salida de propósito general. Los modos de arranque que tenemos disponibles están recogidos en la tabla 3.1.2.1

Evidentemente, el modo de arranque que nos interesa es el primero, por lo que debemos configurar los pines Mode A, Mode B y Mode C a 0. Para esto basta con conectarlos a tierra a través de una resistencia de 10KΩ.

Mode C	Mode B	Mode A	Método de arranque
0	0	0	Se utiliza BDMA para cargar las 32 primeras words del programa de memoria a partir de la memoria de byte. La ejecución se retiene hasta que se han cargado las 32 palabras. El chip queda configurado en modo Full Memory.
0	1	0	No hay arranque automático. La ejecución del programa comienza en la localización 0 de la memoria externa. El chip se configura en modo Full Memory. Se puede seguir usando BDMA, pero el procesador no usa o espera estas operaciones automáticamente
1	0	0	Se utiliza BDMA para cargar las 32 primeras words de el espacio de memoria de byte. La ejecución del programa se retiene hasta que las 32 palabras hayan sido cargadas. El chip se configura en modo Host. Se requiere hardware adicional.
1	0	1	Se utiliza IDMA para cargar el programa interno que se desee. La ejecución del programa se retiene hasta que se escriba en la localización de memoria 0. El chip se configura en modo Host.

Tabla 3.1.2.1: Configuración en el arranque del ADSP-2186L

3.1.2.6. Puertos serie

El ADSP-2186L incorpora dos puertos serie síncronos completos (SPORT0 y SPORT1) para comunicaciones en serie y comunicación multiprocesador.

Estos puertos serie tienen las siguientes características principales:

- Son bidireccionales y tienen secciones separadas de recepción y transmisión con buffers independientes
- Pueden usar un reloj externo o generar su propia señal de reloj internamente

- Tienen entramado independiente para las secciones de transmisión y recepción. Se puede no usar sincronización de trama o bien usar una sincronización de trama generada interna o externamente. Las señales de sincronización de trama están activas a nivel alto o invertidas, con dos posibilidades de anchura de pulso y temporización
- Se soporta la transmisión de palabras de datos con longitudes comprendidas de 3 a 16 bits, y opcionalmente disponemos de compresión / expansión mediante las leyes A o μ según la recomendación G.711 de CCITT.
- Las secciones de transmisión y recepción pueden generar interrupciones independientes al completar la transferencia de una palabra
- Se puede transmitir o recibir un buffer circular completo de datos robando solo un ciclo de reloj por cada palabra de datos, generandose una interrupción tras recibir el buffer.
- SPORT0 tiene un interfaz multicanal para recibir o transmitir selectivamente palabras de 24 o 32 bits en un flujo de bits multiplexado por división en el tiempo.
- SPORT1 puede ser deshabilitado para disponer de 2 interrupciones externas adicionales y banderas de entrada y salida. El reloj del puerto serie generado internamente puede seguir usandose en esta configuración.

Dado que no vamos a usar las características especiales de los puertos, estas no tienen importancia para nosotros, por que no se va a utilizar el interfaz multicanal, ni vamos a prescindir de SPORT1 dado que necesitamos ambos puertos. La selección de que puerto vamos a usar para comunicarnos con el PC y cual vamos a usar para comunicarnos con la otra placa, depende por lo tanto de otros factores.

Viendo las prioridades de las tabla de interrupciones, observamos como el puerto serie 0 tiene una prioridad mas alta que el puerto 1. Debido a esto, usaremos el puerto 0 para comunicarnos con el PC, y reservaremos el puerto 1 para comunicarnos con la placa, por que la comunicación con el PC va a realizarse a una velocidad superior que la comunicación entre placas (de no hacerse así, la compresión no tendría ningún sentido, ya que estaríamos limitados por la velocidad de comunicación PC-placa).

Un problema importante que se nos plantea es que los puertos son síncronos, mientras que los puertos serie del PC son asíncronos. Al analizar la configuración de los puertos

serie de los chips de la familia 21xx de Analog Devices, vemos que no existe opción de configuración para funcionar de forma asíncrona compatible con el PC.

Para resolver el problema, en la dirección ADSP-2186L > PC no tenemos mas que fijar el tamaño de palabra a 10 bits, y en lugar de enviar palabras de 8 bits, se envían de 10 bits incluyendo un bit a 0 al comienzo (el bit de arranque) y un bit a 1 al final (el bit de stop), configurando el puerto serie sin paridad y con un solo bit de parada en el lado del PC.

En la dirección PC > ADSP-2186L, se puede solucionar haciendo que los datos que vienen no solo entren en el pin DR0, sino también en el pin RFS0, que recibe la sincronización de trama, configurando dicha configuración de trama de forma que para cada palabra recibida se active dicha sincronización a nivel bajo y se ignore hasta recibir todos los bits de los que consta la palabra. De esta manera, el bit de arranque que viene del PC actúa como sincronización de trama.

El puerto serie 1 no tiene ningún problema, ya que ambas placas se comunicarán de forma síncrona utilizando las señales de sincronización de trama además de las señales de envío de datos. Por lo tanto, las señales que se envían son DT1 y TFS1, recibiendose DR1 y RFS1. Por su parte, el puerto 0 envía DT0 y recibe DR0 y RFS0 de la misma línea de datos del PC. Aparte, para comunicarnos con el PC usaremos dos de las señales de propósito general, concretamente PF4 y PF7 para establecer el protocolo de handshaking CTS/ RTS.

Para adaptar los niveles de tensión RS-232 con los niveles de alimentación de la placa (3.3V – 0V), se utilizan dos chips MAX3232, cuyo esquema aparece en la figura 3.1.2.2

Los valores de los condensadores C1, C2, C3 y C4 dependen de la tensión de alimentación del circuito, y para Vcc = 3.3V, un valor válido es 0.1μF. Se pueden utilizar capacidades tanto polarizadas como sin polarizar.

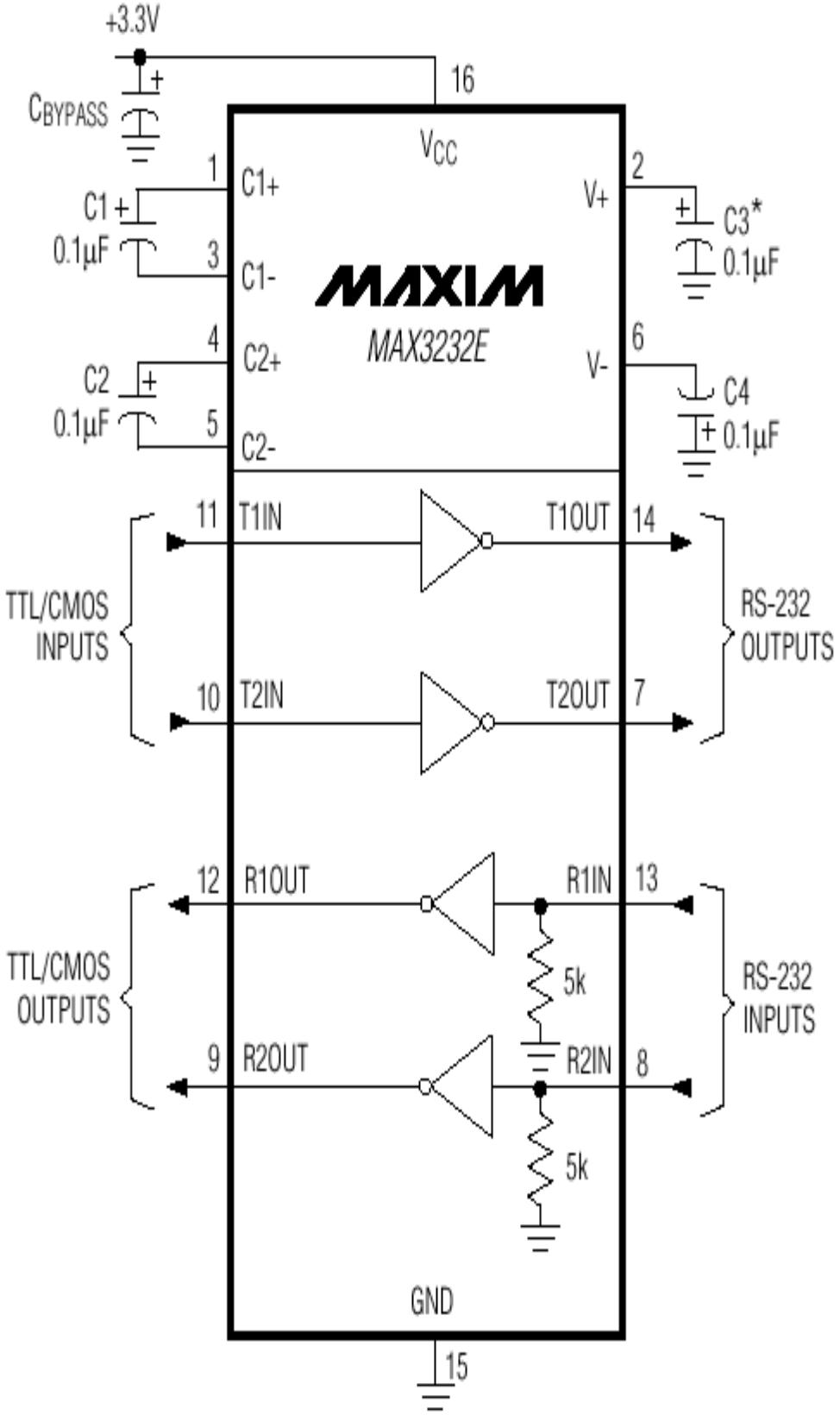


Figura 3.1.2.2: Esquema del chip MAX3232

La conexión de los puertos serie con los chips MAX3232 se realiza a través de los pines 2 y 3 para envío y recepción de datos, 5 para tierra y 7 y 8 para Handshaking CTS/RTS en el caso del SPORT0 y para las señales de sincronización de trama en el puerto SPORT1, aunque en este caso podríamos haber usado también la pareja de pines 4 y 6.

3.1.2.7. Otras conexiones

Una vez que hemos conectado la EPROM, la alimentación, el reset, las líneas de configuración, la entrada del reloj y los puertos serie, tan solo quedan por conectar los LEDs, cuya conexión consiste únicamente en el propio LED seguido de una resistencia de 330 Ω a tierra y algún que otro pin que no deba quedarse sin conexión, como es el caso del pin BR, que debe conectarse a tensión positiva con una resistencia (que se la pondremos de 10 KΩ) en caso de no utilizarse, o el pin PF3, que actúa como pin de configuración en otros modelos de la familia 218x, como es el caso del 2187L, donde actúa de Mode D. Este pin, al igual que los anteriores pines de configuración, lo fijamos a 0 a través de una resistencia de 10 KΩ

3.1.2.8. Creación de la placa

Una vez diseñada la placa, hay que realizar el diseño del rutado y conexión de los componentes utilizando un programa de PCB. La primera placa se diseño usando el programa Accel Tango PCB y el diseño aparece representado en el plano 5.1.

El chip ADSP-2186L es un chip superficial, por eso casi todas las conexiones están el la capa alta (Top). Se ha procurado en todo momento disminuir el número de pads, así como reducir la superficie del circuito.

Las conexiones de la capa Top que sobrepasan el punto de llegada en la EPROM y los MAX3232 estaban pensadas para solucionar el problema que supone hacer las conexiones por la parte de arriba de un conector tipo DIP, pero estos problemas no existen si en lugar de utilizar un conector DIP se utilizan hileras de pines.

A la hora de soldar la placa, tuve algunos problemas por que los pads están muy cerca unos de otros, por que los componentes quedan demasiado pegados en la parte de la alimentación del circuito (impidiendo la colocación de bornas para entrada y salida, por lo que tuvimos que usar cables para alimentarlo), y sobre fueron problemáticos los pads situados dentro del área correspondiente al procesador, dado que este es superficial y se sitúa a una altura muy pequeña sobre la placa, por lo que hay que tener mucho cuidado con las conexiones y procurar hacerlas lo mas planas posibles. Esas 6 conexiones internas son necesarias debido al alto número de tierras y puntos de alimentación que tiene el DSP (concretamente 10 tierras y 6 alimentaciones). En el proceso del soldadura, una de estas conexiones quedó inutilizada por que se perdió el cobre y hubo que apañar la conexión con un cable situado en la parte de debajo de la placa (como puede apreciarse en la Fotografía 2.2.2).

Una vez resueltos los problemas, el siguiente paso era probar el primer programa desarrollado con VisualDSP en ensamblador para ver si el circuito funcionaba correctamente, prueba que transcurrió satisfactoriamente, por lo que se dio por acabado el trabajo con esta placa.

El diseño previsto para la segunda placa (que no llegó a realizarse), evita las conexiones que salen del punto de llegada (pues usando hileras de pines no son necesarias), separa lo máximo posible los pads unos de otros y saca fuera del área del procesador 3 de las 6 conexiones internas, aprovechando el espacio restante en las esquinas. Este diseño (versión 1.1) puede observarse en el plano 5.2 .

3.1.3. MODELO DE PROGRAMACIÓN

3.1.3.1. La familia ADSP-21xx

Desde el punto de vista de la programación, los procesadores ADSP-21xx poseen tres unidades computacionales, dos generadores de direcciones de datos y un secuenciador de programa, además de periféricos dentro del chip y una cantidad de memoria interna que varía con cada procesador. Casi todas las operaciones que usan estos componentes de la arquitectura utilizan uno o mas registros. Los registros internos almacenan datos, direcciones, información de control o información de estado.

Hay dos tipos de accesos para los registros. Los registros dedicados pueden ser leídos y escritos directamente en lenguaje ensamblador, mientras que los registros mapeados en memoria se acceden leyendo y escribiendo en sus correspondientes localizaciones en memoria.

Los registros de la familia se muestran en la siguiente figura 3.1.3.1

3.1.3.1.1. Generadores de direcciones de datos: Data Address Generators

DAG1 y DAG2 tienen cada uno 12 registros de 14 bits: cuatro índices (I) para guardar punteros, 4 registros de modificación (M) para actualizar los punteros y 4 registros de longitud (L) para implementar buffers circulares. DAG1 solo accede a memoria de datos y tiene capacidad para invertir los bits de sus salidas. DAG2 puede direccionar tanto memoria de datos como de programa, y puede proporcionar direcciones para bifurcaciones indirectas además de acceder a datos.

3.1.3.1.2. El secuenciador de programa

Los registros asociados con el secuenciador de programa controlan las subrutinas, bucles e interrupciones. También indican el estado y seleccionan modos de operación.

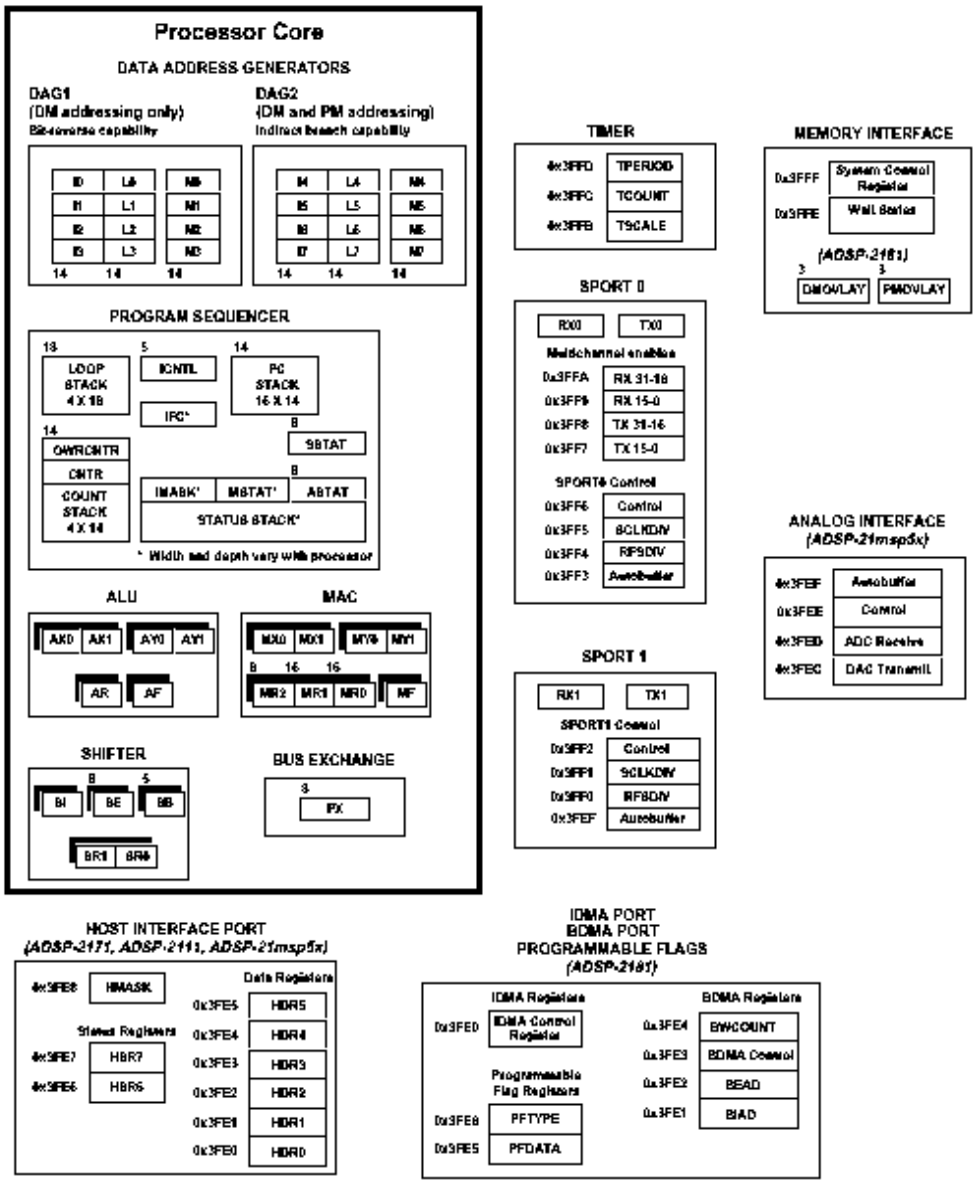


Figura 3.1.3.1: Registros de los DSPs de la familia 21xx de Analog Devices.

Respecto a las interrupciones, el registro ICNTL controla el anidamiento de interrupciones y la sensibilidad a interrupciones externas, el registro IFC permite forzar y limpiar interrupciones por software, e IMASK enmascara interrupciones individualmente. Las anchuras de IFC e IMASK dependen de cada procesador.

Los modelos mas avanzados permiten activar o desactivar las interrupciones globalmente mediante instrucciones en ensamblador. Las interrupciones están activas por defecto al resetear. Al desactivar las interrupciones no se altera el valor de IMASK, por lo que al activar las interrupciones de nuevo las interrupciones no enmascaradas pueden ser atendidas.

Respecto a los bucles, disponemos del registro CNTR que lleva la cuenta del valor del bucle que se está ejecutando actualmente.

El registro de estado de las pilas (SSTAT) contiene banderas de lleno o vacío para las pilas. El registro de estado aritmético (ASTAT) contiene banderas de estado para las unidades computacionales, y el registro de estado de modo (MSTAT) contiene bits de control para varios propósitos.

El secuenciador de programa tiene cuatro pilas que permiten bucles y anidamiento de subrutinas e interrupciones.

1- La pila del PC (Program Counter) tiene 14 bits de anchura y una profundidad de 16 posiciones. Almacena direcciones de retorno para subrutinas e rutinas de servicio de interrupciones.

2- La pila de bucles tiene 18 bits de anchura, 14 bits para la dirección final del bucle y 4 bits para determinar la condición de final, y tiene 4 niveles de profundidad.

3- La pila de estado, en la que se introducen valores automáticamente al atender una interrupción, acomoda los registros IMASK, MSTAT y ASTAT. La profundidad y anchura de esta pila depende de cada procesador. Cuando acaba el servicio de interrupción los valores guardados se restauran automáticamente.

4- La pila de cuenta, tiene 14 bits de anchura y guarda el valor del contador (CNTR) para el anidamiento de bucles basados en contador.

Las pilas de PC y de estado permiten introducir valores manualmente, y todas las pilas permiten sacar valores manualmente.

3.1.3.1.3. Unidades Computacionales

Los registros de las unidades computacionales almacenan datos. La Unidad Aritmético-Lógica (ALU) y el Multiplicador-Acumulador (MAC) requieren dos entradas para la mayor parte de las operaciones. Los registros AX0, AX1, MX0 y MX1 almacenan entradas X mientras que AY0, AY1, MY0 y MY1 contienen las entradas Y.

AR y AF almacenen los resultados de la ALU, pudiendo AF ser usado para retroalimentar la entrada Y de la ALU, mientras que AR puede contener la entrada X de cualquier unidad computacional. De la misma forma, los registros MR0, MR1, MR2 y MF almacenan resultados del MAC y pueden ser retroalimentados para otros cálculos. Los registros de 16 bits MR0 y MR1 junto al registro de 8 bits MR2 pueden almacenar un resultado de multiplicación/acumulación de 40 bits.

El Desplazador (Shifter) puede recibir entradas de la ALU o el MAC, de sus propios registros de resultados o de su propio registro de entrada (SI). Puede almacenar un resultado de 32 bits usando los registros SR0 y SR1. El registro SB almacena el exponente del bloque para las operaciones de punto flotante en bloque. El registro SE almacena el valor de desplazamiento para las operaciones de normalización y desnormalización.

Los registros de las unidades computacionales están duplicados, lo cual es útil para cambiar el contexto en un solo ciclo de reloj. La selección del juego de registros primario o secundario se controla mediante un bit del registro MSTAT.

3.1.3.1.4. Intercambio de bus

El registro PX es un registro de 8 bits que permite transferir datos entre el bus de 16 bits de memoria de datos y el bus de 24 bits de memoria de programa. En una transferencia entre estos buses, PX provee o recibe los 8 bits mas bajos.

3.1.3.1.5. Temporizador (Timer)

Los registros mapeados en memoria TPERIOD, TCOUNT y TSCALE almacenan respectivamente el periodo, la cuenta y el factor de escala del temporizador.

3.1.3.1.6. Puertos Serie

SPORT0 y SPORT1 tienen cada uno sus propio registros de recepción (RX), transmisión (TX) y control. SPORT0 tiene además registros para controlar sus capacidades multicanal. Cada registro de control tiene bits dedicados a la sincronización de trama, compresión, longitud de palabra y en el caso del SPORT0, las funciones multicanal. Para cada puerto serie el registro SCLKDIV determina la frecuencia del reloj serie generado internamente, y el registro RFSDIV determina la frecuencia de la señal de sincronismo de trama generada internamente. Los registros de autobuffer controlan dicha propiedad en cada puerto.

3.1.3.1.7. Interfaz de memoria y activación de puertos serie

El registro de Control de Sistema (Control System Register) contiene la activación de los puertos serie así como la configuración del puerto serie 1, así como un campo que especifica cuantos estados de espera debe haber al acceder a la memoria de programa externa.

Por último, el registro de Control de Estados de Espera (Wait State Control) contiene los estados de espera para cada banco de memoria de datos, y en los ADSP-218x también especifica el número de estados de espera para la memoria de E/S.

Consideraciones sobre la programación

Aunque lo máximo que se llegó a desarrollar fue un ejemplo para encender y apagar los LEDs a intervalos regulares usando el timer, sin llegar a escribir el codec de compresión, hay ciertas ideas que fueron tenidas en cuenta antes de dar por terminado el diseño de la parte hardware:

- La capacidad de autobuffer permite cargar un determinado número de bytes del puerto serie sin tener que estar pendiente a la recepción de estos uno a uno, lo cual significa que tendremos mas tiempo disponible para dedicarlo a la compresión.
- Los registros L de los generadores de direcciones posibilitan el empleo de buffers circulares de cualquier tamaño, lo cual viene muy bien para desarrollar el algoritmo de compresión.
- La solución dada al problema de la comunicación serie con el PC de forma asíncrona parece funcionar teóricamente, pero habría que probarla para ver hasta que punto es viable.
- Hay que estar atento al estado del buffer de salida, por que si este se queda vacío, es mas conveniente transmitir un cierto número de bytes sin comprimir que dejar pasar el tiempo sin estar enviando nada.

3.2. SOFTWARE

La parte del proyecto software consta en primer lugar de un programa de comunicaciones desarrollado en Visual Basic que se encarga de todo lo referente a las comunicaciones por el puerto serie, y desde el cual se escogen los archivos que se quieren transmitir y con que nombre y en que directorio van a almacenarse aquellos que nos lleguen, y en segundo lugar de unos codecs de compresión y descompresión desarrollados en ensamblador que aplican las técnicas de compresión que habíamos diseñado en un principio para la solución Hardware, aunque para dicha solución no se hayan llegado a programar los algoritmos en código nativo de ADSP-218x.

La forma de comunicación que tienen los codecs con el programa principal en Visual Basic es mediante un archivo de entrada, en el cual aparece el nombre del archivo con el que van a tratar. Esto se ha hecho así por que la otra solución posible (pasar dicho nombre de archivo como argumento para el codec), tiene el problema de la limitación de dicha línea de comandos a un máximo de 127 caracteres, mientras que un nombre largo de archivo junto a su path y unidad puede ocupar hasta 260 caracteres en Windows, por lo que los archivos con nombres muy largos o presentes en directorios con nombres largos o muy profundos podrían dar problemas.

Debido a esto, los codecs de compresión buscan el nombre, unidad y path del archivo de entrada (el que va a ser comprimido) en un archivo llamado “ENTRADA.ARC”, y crean un archivo de salida con el nombre “SALIDA.CMP”, que es posteriormente enviado por el programa principal. De forma similar, los codecs de descompresión leen su entrada de un archivo llamado “ENTRADA.CMP” (que es el que se ha recibido por el puerto serie) y escribe su salida creando un archivo cuyo nombre, unidad y path vienen indicados en el archivo “SALIDA.ARC”.

3.2.1. ANÁLISIS DEL CODEC DE COPIA

Uno de los métodos de compresión disponibles en el programa principal es el llamado “Sin compresión”. En este caso, no haría falta que tuviesemos un codec específico para este método, ya que el archivo puede leerse desde el programa principal directamente y también escribirse, pero el codec de copia sirve para comprobar que el mecanismo de selección de archivos y el esquema de comunicación del programa principal con sus codecs funciona correctamente.

El codec de copia es al igual que los demás codecs, un programa .COM de MSDOS, que utiliza por lo tanto funciones de la API del sistema operativo invocando la interrupción 21h con el número de función en AH y los parámetros de las funciones en registros, en lugar de utilizar llamadas a funciones de la API de Win32 colocando los parámetros de llamada en la pila, como hacen los programas nativos de Windows. Esto no obstante no supone ningún problema a la hora de operar con archivos con nombres largos, ya que con Windows 95 se añadieron funciones para operar con nombres largos en la API del COMMAND.COM (que es la que se invoca con int 21h).

3.2.1.1. COPYPA.COM

Lo primero que hace este programa es llamar a la función 3ch para crear un archivo, con los atributos a 0 y apuntando al nombre "SALIDA.CMP". El número de la función se coloca en AH, los atributos en CX y el puntero al nombre en DX. Una vez que se llama a la función, al volver se indica en el flag C si ha habido algún error, y de no existir ninguno, en AX se devuelve el handle o identificador de archivo, que pasamos a guardar en una variable.

El siguiente paso es abrir el archivo (función 3dh) de nombre "ENTRADA.ARC", tras lo cual y si no ha habido ningún problema se lee dicho archivo colocando su handle en el registro BX, el número de bytes a leer en CX, la función 3fh en AH y la dirección donde escribir los bytes leídos en DX.

Una vez hecho esto, tenemos que usar una función de nombre largo para abrir el archivo de entrada, concretamente la función se especifica con AX=716Ch, el modo de acceso de solo lectura en BX, los atributos en CX y la acción (abrir archivo existente, dado que esta función de nombre largo sirve tanto para abrir archivos como para crear otros nuevos) en DX, mientras que el nombre del archivo se coloca en esta ocasión en SI. Abierto este archivo de entrada, entraríamos en el bucle de copia, que es propio de este codec. Todos los pasos desde el comienzo hasta aquí son idénticos en todos los codecs de compresión.

En cuanto a la copia, no tiene ningún misterio, tan solo se van leyendo bloques de 6000 bytes del archivo de entrada y tal como se leen se van escribiendo al archivo de salida. Para esto se emplean las funciones 3fh (lectura) y 40h (escritura), usando en cada caso su handle correspondiente. Al realizar una lectura, la API devuelve en AX el número de bytes leídos, si este número es 0 saltamos al final del programa, y si es mayor de 0 lo pasamos a CX para que se escriban tantos bytes como se han leído.

Al final del programa, sencillamente salimos al sistema operativo, con lo cual se cierran todos los archivos que teníamos abiertos. En la salida a Windows especificamos un valor de retorno 0, que significa que no ha habido ningún error. Si en alguna llamada

a la API aparece un error (o sea, se devuelve el flag de acarreo activado), salimos especificando un errorlevel diferente, que depende de la situación específica del error, pero desgraciadamente la orden Shell de Visual Basic no devuelve el valor de errorlevel con el que sale el programa, por lo que este tratamiento de errores es superfluo.

3.2.1.2. COPYUN.COM

Este programa es prácticamente idéntico al anterior, la diferencia está en que comienza abriendo el archivo ENTRADA.CMP, y tras obtener el nombre largo del archivo SALIDA.ARC, tiene que crear el archivo de salida utilizando la función de nombre largo, para lo cual ha de especificar una orden distinta en DX, de tal forma que si el archivo existe lo trunque, es decir, desecha su contenido, y si no existe lo crea.

Este es el único codec de descompresión que es simétrico a su equivalente de compresión, por razones obvias.

3.2.2. ANÁLISIS DE LOS CODECS LZ

3.2.2.1. Diseño

El algoritmo de compresión que he desarrollado es el mismo que tenía pensado utilizar para el ADSP-2186L. La principal limitación de este procesador es la cantidad de memoria disponible, (solo 8K words (de 16 bits) de memoria de datos y otros 8K words (en este caso de 24 bits) de memoria de programa), pero también es un factor decisivo la velocidad del micro (33Mhz, lo cual no es demasiado). Aparte de esto, está el hecho de que va a utilizarse para comprimir en tiempo real, por lo cual necesita ser lo más rápido posible.

En primer lugar lo primero que decidí fue usar una codificación LZ sencilla, sin flexible ni Lazy parsing, para favorecer la velocidad, y sin utilizar un codificador posterior como Huffman o codificación aritmética.

El siguiente asunto a decidir fue el algoritmo de búsqueda, ¿debía utilizar árboles, o por el contrario utilizar fuerza bruta? Ni una cosa ni la otra, lo que pensé fue utilizar un espacio de memoria del mismo tamaño que el diccionario para almacenar la distancia al byte previo mas cercano que tenga el mismo valor que el actual, de forma que no hace falta buscar hacia detrás comprobando el valor de cada byte hasta encontrar un comienzo de coincidencia para después escanear las coincidencias hacia delante, sino que el primer byte coincidente lo tenemos seguro y una vez comprobado cada uno vamos saltando hacia atrás para encontrar el siguiente. Para emplear esto, hace falta además otra tabla de 256 words (una para cada valor posible de byte) en la que se almacena la última posición que contuvo ese byte. Con cada byte nuevo que procesamos, se coge el valor de dicha tabla de últimas apariciones para calcular la distancia respecto a la posición actual, y se actualiza tanto la tabla de saltos previos como la tabla de última aparición del byte en cuestión. Esto se hace con todos los bytes, incluso con aquellos que ya han sido codificados en un codepair y por lo tanto no vamos a buscar cadenas repetidas con ellos. Una vez que estamos buscando las longitudes de las posibles coincidencias para quedarnos con la mayor, vamos acumulando la distancia para evitar salirnos de la ventana deslizante. Dado que el buffer es circular y los valores mas antiguos se van sobrescribiendo, los datos presentes en la tabla de saltos a previos

pierden su significado cuando se sobrescriben los datos, pero esos datos no llegan a utilizarse dado que la distancia máxima nos impide que lleguemos a ellos antes de que los hayamos actualizado. Con la tabla de últimas posiciones no sucede lo mismo, por lo que es posible que la información contenida en dicha tabla no sea válida, lo cual se comprueba viendo si el byte contenido en la dirección indicada se corresponde efectivamente con el que pretendemos encontrar. Si esto no es así, o si es la primera vez que se lee un byte (y por lo tanto la tabla de última posición está vacía), se genera directamente un byte literal, fijando a la vez un número muy alto (del tamaño del buffer-1, para asegurarnos de que nos pasamos) en la posición de la tabla de saltos a previos. Este método es peor que los arboles sufijos, pero por lo general suele ser bastante mas rápido que la búsqueda por fuerza bruta, aunque en determinados peores casos puede ser contraproducente.

Lo siguiente que hay que escoger es el tamaño del diccionario, y basándome en las limitaciones del DSP y en las necesidades de memoria, decidí usar un diccionario de 6K para transmisión semi-duplex y reducir el tamaño a la mitad para transmisión full-duplex, ya que se necesitan al menos 256 + 2 * T words para la compresión (siendo T el tamaño del diccionario) y T words para descomprimir (ya que para la descompresión no buscamos matches, por que nos vienen ya dados y no necesitamos las tablas de última posición y saltos previos).

Una vez que sabemos el tamaño del diccionario, también sabemos cual es la longitud máxima que puede alcanzar la distancia en un match, y por lo tanto cuantos bytes requiere como máximo. De aquí pasamos a decidir cual va a ser el formato del archivo comprimido: que tipos de códigos va a haber y como se forman.

Respecto al tipo de códigos, he decidido que va a haber literales simples, codepairs y cadenas de literales, esto último son un número de literales que van todos agrupados de forma que no necesitan un flag para cada uno de ellos. La inclusión de este tipo de código viene motivada por la conveniencia de minimizar el incremento de tamaño de los archivos que no pueden comprimirse. Si no se encuentra ningún match, con un esquema tradicional LZ77 a base de codepairs y literales, todos los bytes serían literales, por lo que por cada 8 bits se emitirían 9 (los 8 del literal y el de su bandera), y los archivos no comprimibles aumentarían por tanto su tamaño en un 11%. Al usar cadenas

de literales, al sobrepasar un cierto número de literales que van seguidos se juntan todos en un solo código que indica cuantos vienen, por lo que las pérdidas se reducen considerablemente. Por ejemplo, si utilizamos 10 bits por cada 250 bytes literales que se emiten, el aumento del tamaño provocado por la compresión fallida será solo del 0.5%.

Respecto a la codificación de las longitudes y distancias de los codepairs, he decidido usar para las distancias 4 bits que indican cuantos bits extra tiene la distancia, de forma que se codifica como XXXXx…x, siendo el número de bits x…x el indicado por XXXX, y la longitud que se codifica es 2^X+xxxx (es decir, el primer bit del número binario está implícito). Para las longitudes, tanto las correspondientes a los codepairs como a las cadenas de literales, lo mismo pero usando 3 bits (o sea, YYYy…y).

Un primer bit a 0 indica un literal simple, mientras que un 1 puede indicar varias cosas: una cadena de literales, un codepair o el código de final de archivo.

Para las longitudes podemos expresar valores desde 1 (000) hasta 255 (1111111111). En el caso de los codepairs, diremos que si YYY =111, hemos alcanzado el final del archivo, con lo cual el rango de longitudes que se pueden expresar para un codepair es de 1 a 127, que al sumarle 1 dado que la longitud mínima es 2, queda en 2-128. Esto hace que debamos tener leídos al menos 128 bytes mas que el que está siendo procesado actualmente. Por lo tanto la distancia máxima que podemos retroceder es de 6K-128 bytes = 6016, pero esto será solo en el caso extremo en que solo tengamos 128 bytes pendientes en el buffer, ya que enseguida leeremos otros 128 bytes y la distancia máxima oscilará entre 5888 y 6015 bytes. Esto puede expresarse con 13 bits, lo cual indica que XXXX debe estar comprendido entre 0000 y 1100. A partir de 1101, estaremos en el caso de cadena de literales, en el cual los dos últimos bytes de XXXX se corresponden con los dos primeros de YYY, y por lo tanto YYY está comprendido entre 010 (con lo cual el mínimo número que puede representarse es el 01000 > 100, 4) y 111. Nos interesa que el menor número de literales de una cadena (código 11101000, 8 bits) represente un número de bytes de forma que obtengamos algún beneficio al usar la cadena de bytes, por lo que sumaremos 4 a la cifra obtenida y los literales transmitidos oscilarán entre 8 y 259.

Una última consideración de diseño es que la longitud mínima de coincidencia, que es 2, solo se tendrá en cuenta si la distancia es menor que 256, ya que en caso contrario es mas conveniente sacar literales.

En el caso del diccionario de 3K, la distancia máxima puede expresarse con 1 bit menos, por lo que XXXX variará de 0000 a 1011, y por lo tanto YYY puede expresar cualquier cifra del 1 al 255, y el código mínimo (111000) son 6 bits, por lo que podemos expresar cadenas de bits desde 7 a 261 bytes de longitud, por lo demás la codificación queda idéntica.

La última decisión que nos queda por tomar es respecto a usar el esquema de flujo continuo de bits o bien diferenciar entre tag-bits y literales. A favor de la eficiencia, escogí lo segundo, aunque esto implica una mayor complejidad de codificación (especialmente ahora que hay que tener en cuenta el número de literales que van seguidos antes de emitir ningún código para ellos, ya que no sabemos a priori si van a ir sueltos o dentro de una cadena de literales).

3.2.2.2. Detalles a tener en cuenta

Los tag-bits (todos los bits que no corresponden a bytes literales) se van colocando uno a uno en un byte hasta que este byte queda relleno. La manera de detectar esto es colocar el llamado “marker-bit” (bit de marca) en el bit 0 del tag_byte, antes de introducir ningún tag-bit. Este bit de marca se irá desplazando hacia la izquierda conforme vayamos introduciendo mas bits, y una vez que hayamos completado con los 8 bits que necesitamos, el bit de marca saldrá del byte y se excitará el flag de acarreo. Esta técnica del marker bit es una de las ventajas de programar en ensamblador, ya que el lenguaje C no tiene control directo sobre esta bandera y para hacer introducir bits uno a uno en un byte necesita utilizar una variable adicional que nos indique por que bit vamos o bien utilizar una variable de mayor anchura y detectar el acarreo de byte usando una comparación para comprobar si nos pasamos de 256.

De cualquier forma, una vez emitido el tag_byte, tenemos que colocar a continuación todos los bytes literales que tenemos pendientes, por que es así como los espera la rutina de descompresión. Para esto, los bytes literales tienen dos variables que dicen cuantos literales están a la espera para ser enviados y donde comienzan.

Por otra parte, al codificar un literal, no podemos pasarlo directamente a la lista de espera y emitir su código, ya que tenemos que esperar para ver cuantos literales existen seguidos y actuar en consecuencia. Por lo tanto, necesitamos una variable que nos diga cuantos literales llevamos acumulados hasta el momento, y desde donde comienzan a acumularse. Estos literales serán codificados cuando se de uno de estos dos casos:

1 – Hemos encontrado una coincidencia (match):

En este caso decidiremos si emitimos tag bits de literales sueltos o los correspondientes a una cadena, dependiendo del número de literales que se han acumulado.

2 – Hemos alcanzado el máximo número de literales que se pueden codificar en la misma cadena de literales:

En este caso no hay ninguna decisión que tomar, sencillamente se emiten los tag-bits, que corresponden a una cadena de literales de longitud máxima.

De cualquier forma, si emitimos los tag bits sueltos, tendremos que ir uno a uno incrementando los literales que están a disposición de ser emitidos, incrementando la posición de comienzo de los bits sin codificar y decrementando el número de literales sin codificar antes de emitir cada bit 0, ya que si los sumasemos todos a la vez y cuando estamos transmitiendo los tag bits llegamos al final del tag byte actual, emitiríamos mas literales de la cuenta.

Similarmente, si emitimos una cadena tenemos que colocar todos los tag bits de la cadena menos 1, sumar todos los literales sin codificar a la variable de literales listos para emitir y entonces colocar el último tag bit, para que si alcanzamos (lo cual es muy probable) el final del tag-byte cuando no hemos emitido el código completo no pasemos a emitir los literales directamente.

Los buffers son circulares, y no se realiza un movimiento de datos, por lo que tenemos que detectar cuando hemos llegado al final del buffer con un puntero para poder volver para atrás. Esto el DSP no sería necesario por que se pueden crear buffers circulares de cualquier tamaño que se gestionan automáticamente. Para evitar estas comprobaciones

de límite al buscar coincidencias, se hace una copia de los 128 primeros bytes del buffer al final del mismo.

3.2.2.3. Variables utilizadas

buffer > buffer de 6Kb que actúa a la vez de buffer del archivo de entrada y diccionario (ventana deslizante) para el algoritmo LZ.

prevbuff > buffer de 6K words (12K bytes) que se utiliza para guardar la distancia hasta la posición anterior en el buffer del mismo byte que el contenido en el buffer.

outbuffer > buffer de 32K bytes para la salida

prev > tabla de 256 words que contiene la última aparición del byte correspondiente a la posición de la tabla

EOF > indica la posición del final del archivo. Si no está apuntando dentro del buffer, es que aún no hemos visto dicho final de archivo.

current > posición de memoria siendo procesada

tagged_literal_start > posición de comienzo de los literales ya marcados

tagged_literals > número de literales marcados

untagged_literal_start > posición de comienzo de los literales ya marcados

untagged_literals > número de literales sin marcar

tag_bits > tag byte en construcción

codepaired > número de bytes que no tienen que ser procesados por que ya han sido codificados en un codepair

currentoutput > puntero a la posición actual en el buffer de salida

maxdistance > hasta cuanto podemos retroceder al buscar un match

readedge > límite de proceso hasta que volvamos a leer un bloque de 128 bytes.

handlein > identificador del archivo de entrada

handleout > identificador del archivo de salida

3.2.2.4. Pseudocódigo

Obviando el trato con archivos, el mecanismo de búsqueda de coincidencias es mas o menos el siguiente (pseudocódigo parecido al basic, las constantes se han fijado para el caso del diccionario de 6K):

```
maxdistance = -1
EOF = 65535
current = buffer
tag_bits = 1
tagged_literal_start=tagged_literals=untagged_literal_start=untagged_literals=0
readedge = buffer + 6016
```

bucle:

```
maxdistance = maxdistance + 1
if current = EOF then goto final
currbyte = buffer[current]
current = current + 1
if current = readedge then
        leer 128 bytes a partir de readedge + 128
        if (se leen menos de 128 bytes)
                colocar EOF
```

```
                si no
                        actualizar readedge
                end if
                maxdistance = maxdistance - 128
        end if
        currlast = prev(currbyte)
        prev(currbyte)=current
        distance = current - currlast
        prevbuff(current) = distance
        if codepaired>1 then
                codepaired = codepaired - 1
                goto bucle
        end if
        if currlast = 0 then goto setliteral_prevbuff
        if buffer(currlast)<>currbyte then goto setliteral_prevbuff
        max_match_lenght = 0

loop00:
        if distancia > maxdistance then goto exitsearchloop
        position_temp_curr = current
        position_temp_comp = current - distance
        position_comp = current - distance
        match_lenght = 0

keepmatching:
        if buffer(position_temp_curr)=buffer(position_temp_comp) then
                position_temp_curr = position_temp_curr + 1
                position_temp_comp = position_temp_comp + 1
                match_lenght = match_lenght + 1
                if match_lenght = 128 then goto nomore
                if current = EOF then goto nomore
        end if
        goto keepmatching
```

```
nomore:
        if match_lenght > max_match_lenght then
                max_match_lenght = match_lenght
                max_match_distance = distance
                if match_lenght = 128 then goto exitsearchloop
        end if
        distance = distance + prevbuff(position_comp)
        goto loop00

exitsearchloop:

        if max_match_lenght < 2 then goto setliteral
        if max_match_lenght = 2 and max_match_distance > 255 then goto setliteral
        if untagged_literals > 0 then gosub tagliterals
        codepaired = max_match_lengh - 1
        poner código de coincidencia, codificar distancia y longitud
        goto bucle

setliteral_prevbuff:
        prevbuff(current) = 6015
setliteral:
        if (untagged_literals = 0) then untagged_literal_start = current
        untagged_literals = untagged_literals + 1
        if untagged_literals = 261 then call tag_literals
        goto bucle

tagliterals:
        if untagged_literals < 8 then goto plain_literals
        codificar cadena de untagged_literals bytes, excepto el último bit
        if tagged_literals = 0 then tagged_literal_start = untagged_literal_start
        tagged_literals = tagged_literals + untagged_literals
        codificar ultimo bit restante
        untagged_literals = 0
        return
```

```
plain_literals:
        if tagged_literals = 0 then tagged_literal_start = untagged_literal_start
        tagged_literals = tagged_literals + 1
        codificar literal simple (bit a 0)
        untagged_literal_start = untagged_literal_start + 1
        untagged_literals = untagged_literals - 1
        if untagged_literals > 0 then goto plain_literals
        return

final:
        if untagged_literals>0 then gosub tagliterals
        poner código de fin de archivo
        vaciar buffer de escritura
        salir al sistema operativo
```

3.2.3. ANÁLISIS DEL TERMINAL DE COMUNICACIONES

El programa principal de comunicaciones fue desarrollado en Visual Basic, debido a la facilidad de programación de una aplicación Visual para Windows que ofrece este entorno. Se encarga de enviar archivos a través del puerto serie, una vez comprimidos. Para comprimir los archivos, utiliza los codecs de compresión o copia que hemos visto anteriormente. Basta con hacer un par de cambios sencillos para añadir un nuevo compresor, y mediante archivos .BAT podemos incluso hacer que utilice compresores externos, posibilidad que será aprovechada para hacer las pruebas en la siguiente sección.

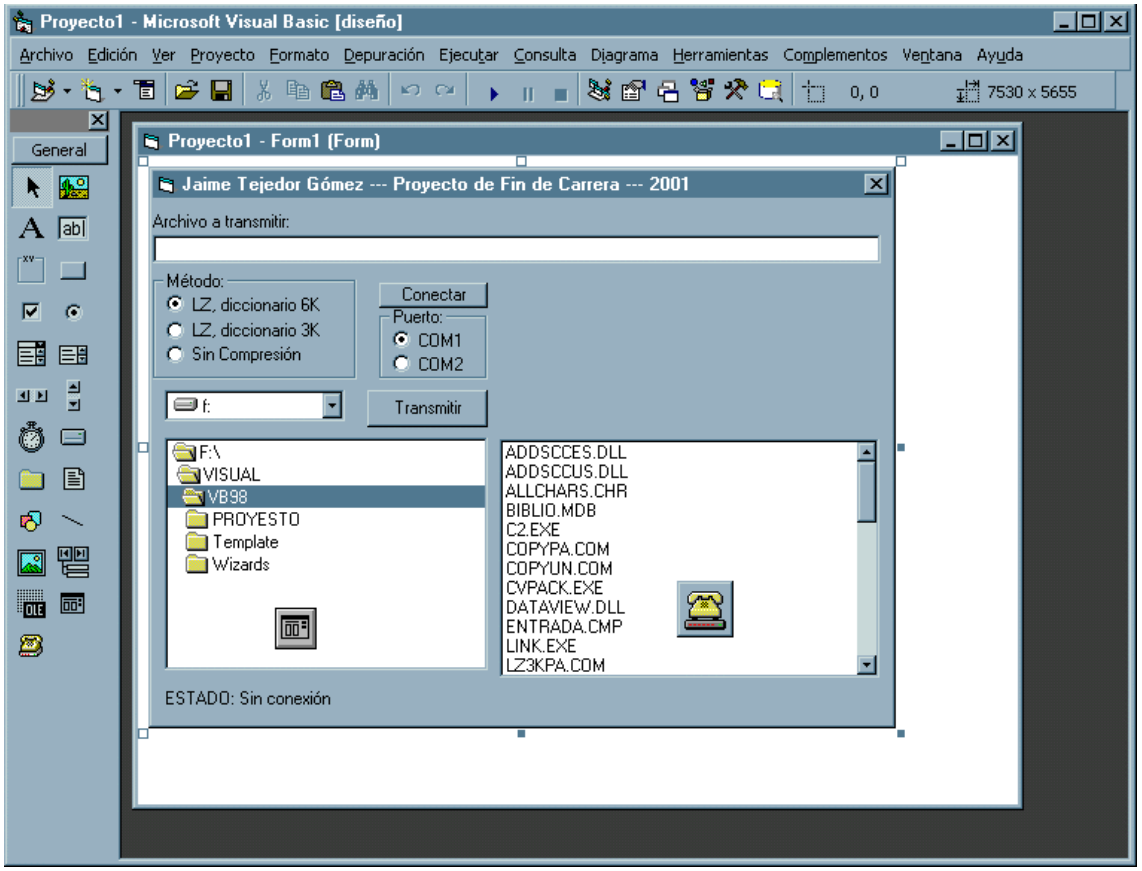
3.2.3.1. Componentes utilizados

En la captura 3.2.3.1, podemos ver el entorno de Visual Basic al desarrollar la aplicación. En la ventana del proyecto se ve la interfaz visual que tiene el proyecto en su único formulario. Hay un cuadro de texto en el que aparece el nombre del archivo que se ha seleccionado para ser transmitido arriba del todo, abajo tenemos un marco con los botones de opción que nos permiten escoger el método de compresión, a su derecha hay otro marco para escoger el puerto serie que vamos a utilizar en la conexión. Encima de este segundo marco hay un botón de pulsación que nos permite abrir y cerrar el puerto serie, y debajo hay otro botón que sirve para enviar un archivo seleccionado. Debajo de todo esto tenemos un selector de unidad de disco, otro de directorio y a la derecha de ambos el selector de archivos.

Sobre el selector del directorio aparece un icono que representa el control OCX CommonDialog, que sirve para utilizar ventanas estándar que son Abrir, Guardar como, Color, Fuente, Imprimir e Invocar Ayuda. En mi caso, solo he empleado el control para abrir una ventana del tipo Guardar Como cada vez que llega un archivo completo y tenemos que escoger donde y con que nombre guardarlo.

Por otra parte, sobre el selector de archivos aparece otro icono, algo mas grande que el anterior y que representa un teléfono, este es el control MSCOMM32.OCX, que se encarga de gestionar toda la comunicación serie, por lo que resulta imprescindible para

el proyecto. Este control no se encuentra disponible en la versión estandar de Visual Basic, por lo que necesitamos la versión profesional o la industrial para compilar el software.



Captura 3.2.3.1. Desarrollo en Visual Basic.

Las propiedades que vamos a usar con este control son:

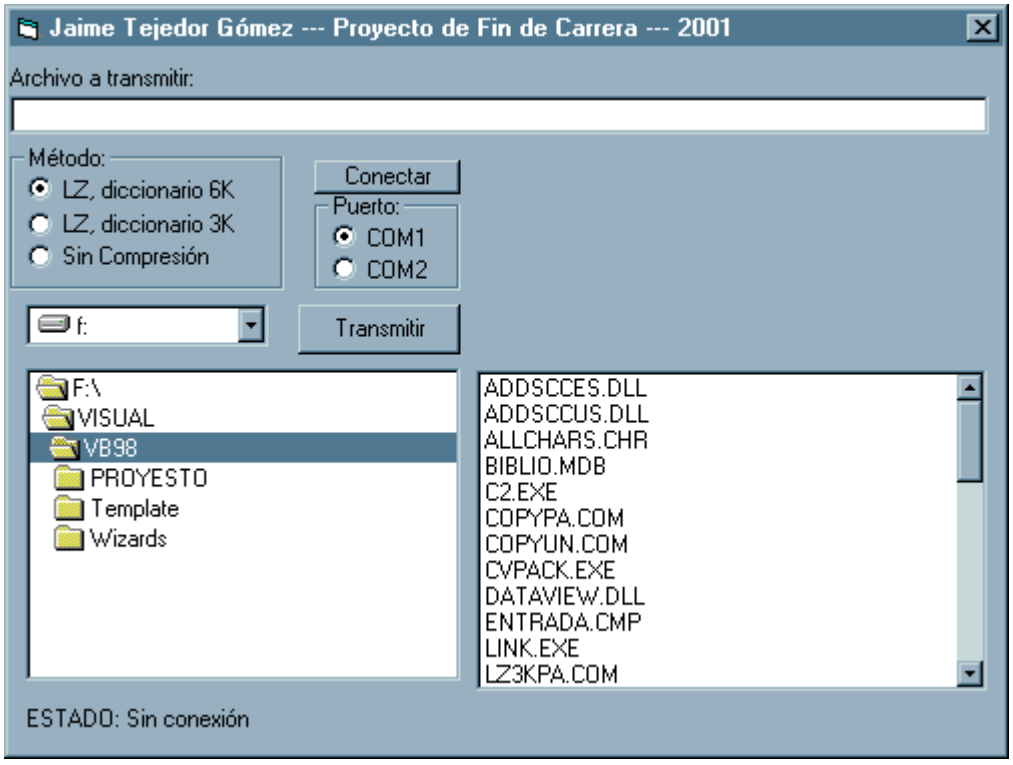
- Settings: 115200,n,8,1 (es decir, transmitir a 115200 bps de velocidad, sin paridad, palabras de 8 bits y con 1 bit de parada).
- Handshaking: 2 – comRTS (al principio usé 0 – comNone, pero daba problemas de sobreescritura al transmitir simultáneamente dos archivos grandes).
- Buffer de entrada de 16384 bytes y de salida de 8192.
- RThreshold y Sthreshold a 1 (mínimo número de caracteres a recibir y enviar antes de emitir un suceso de MSComm1.CommEv
- NullDiscard desactivado (no ignoramos los ceros)
- RTSE y DTRE activados

Por último tenemos etiquetas informativas de texto, no solo la de abajo que informa sobre el estado de la conexión, sino que existen otras 6 etiquetas en la parte de la derecha que parece desierta, que nos informan sobre el tamaño de archivos recibidos y transmitidos, tanto comprimidos como sin comprimir, así como del estado de envío y recepción de los archivos en curso.

3.2.3.2. Funcionamiento del programa

El funcionamiento del programa es muy simple. Necesitamos que el programa se ejecute en dos ordenadores distintos que se comunican mediante un cable null-modem, y en caso de no tener dos ordenadores disponibles, podemos utilizar los dos puertos serie del mismo ordenador (siempre y cuando estén libres) y ejecutar el programa en dos directorios diferentes, cada uno de los programas se conectará con un puerto y no habrá conflictos (si se ejecutan en el mismo directorio van a tener problemas por que tratarán de usar ambos los mismos archivos temporales, a menos que transmitamos solo en semi-duplex, nunca en full-duplex).

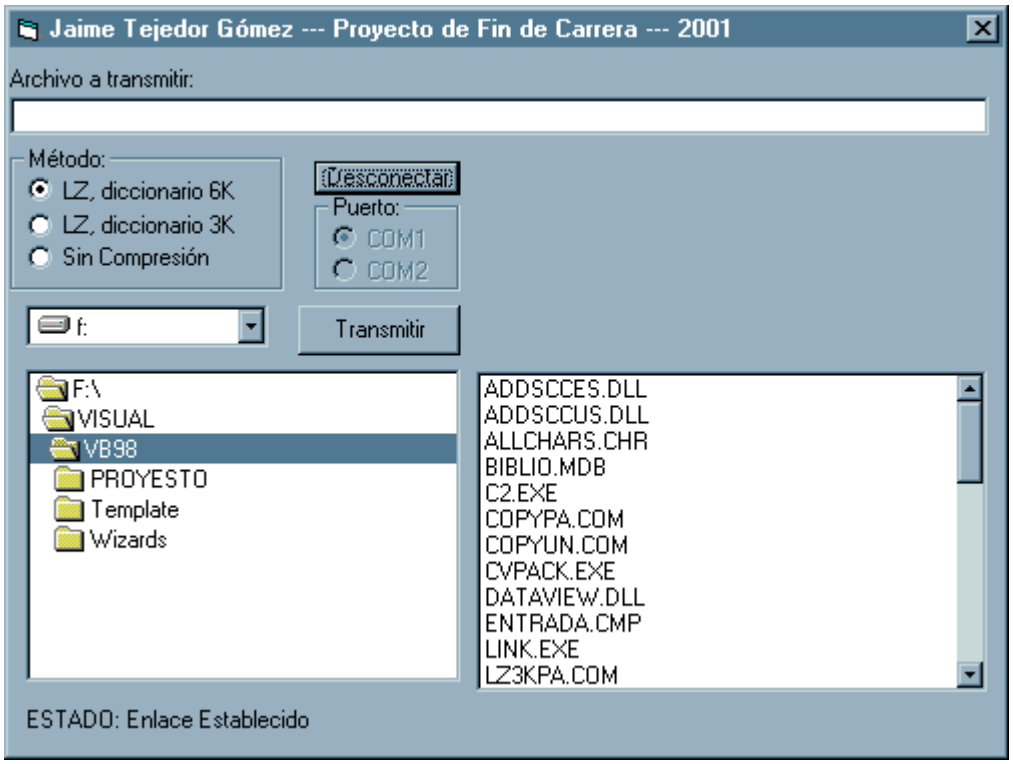
Al ejecutar el programa obtenemos algo así como la captura la captura 3.2.3.2:



Captura 3.2.3.2: Arranque del programa. Sin conexión

Lo primero que debemos hacer es seleccionar el puerto COM al que tenemos conectado el cable y pulsar el botón Conectar. Enseguida cambiará el texto del botón y el estado que aparece abajo del todo pasará a poner: Conectado.

Cuando hagamos lo mismo en el otro lado de la comunicación, el estado cambiará a Enlace Establecido en ambos programas. Esto es debido a que al abrir el puerto mandamos un Ping, de forma que el primer programa que se abre, envía el Ping pero no recibe respuesta por que en el otro lado el segundo no se ha conectado todavía, pero cuando se conecta el segundo, su Ping alcanza al primero, que muestra el mensaje y envía su respuesta al segundo, que al recibir el Pong muestra también el mensaje de Enlace Establecido. Esto se puede ver en la captura 3.2.3.3:



Captura 3.2.3.3: Enlace establecido

Ahora ya podemos seleccionar un archivo cualquiera del directorio actual, o bien navegar por las unidades y directorios para encontrar un archivo que queramos enviar. Para hacer esta prueba escogí un archivo grande y que se puede comprimir muy bien, por que es código fuente muy redundante. Al pulsar sobre el nombre del archivo, su nombre aparece en el cuadro de texto. También podríamos haber escrito directamente el

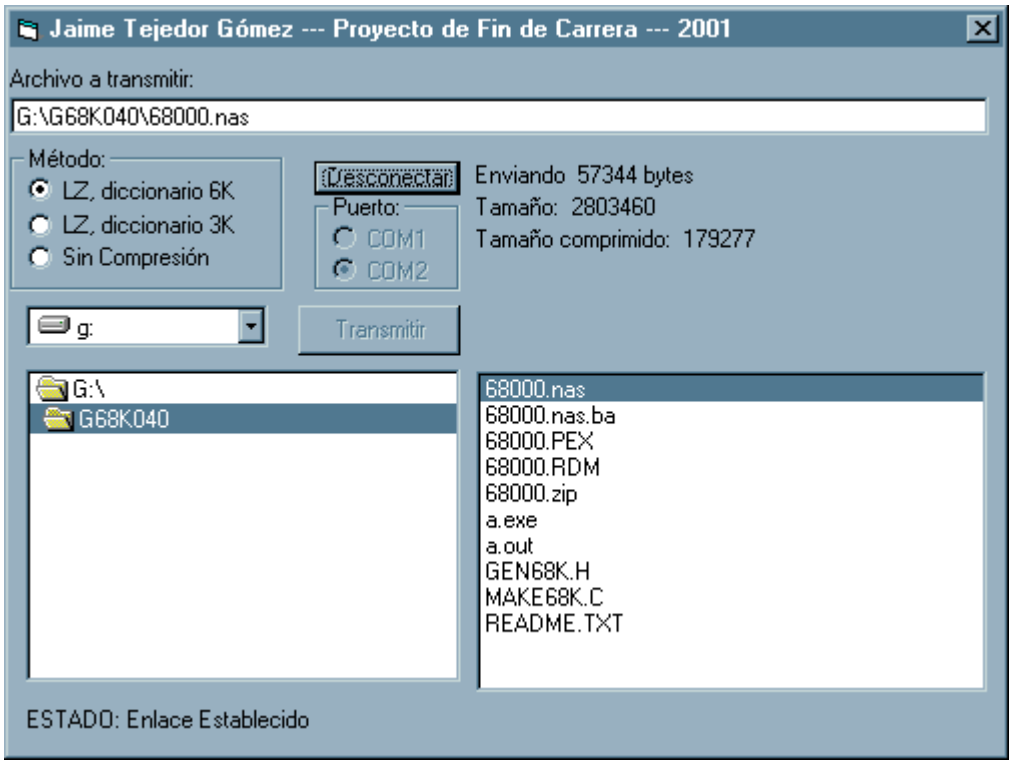
nombre del archivo con su unidad y path sobre el cuadro de texto, pero haciendolo así es posible equivocarse. Vemos este paso en la captura 3.2.3.4:

Captura 3.2.3.4: Escogiendo el archivo

Ahora, nos aseguramos que tenemos escogido el método de compresión que queremos y pulsamos transmitir. En nuestro ejemplo se obtiene la captura 3.2.3.5:

Podemos ver el tamaño original del archivo y mientras se llama al codec en ensamblador que se encarga de comprimir el archivo, aparece arriba a la derecha el texto “Comprimiendo”. El proceso de compresión es bastante rápido, aunque esto por supuesto depende de cada ordenador, por que no será igual de rápido en un Pentium 133 que en un Athlon a 1 GHz. De cualquier forma, en este caso podemos ver el texto por que el archivo es grande, si fuese mas pequeño, apenas veríamos ese texto, sino que pasaríamos a la siguiente, representada en la captura 3.2.3.6:

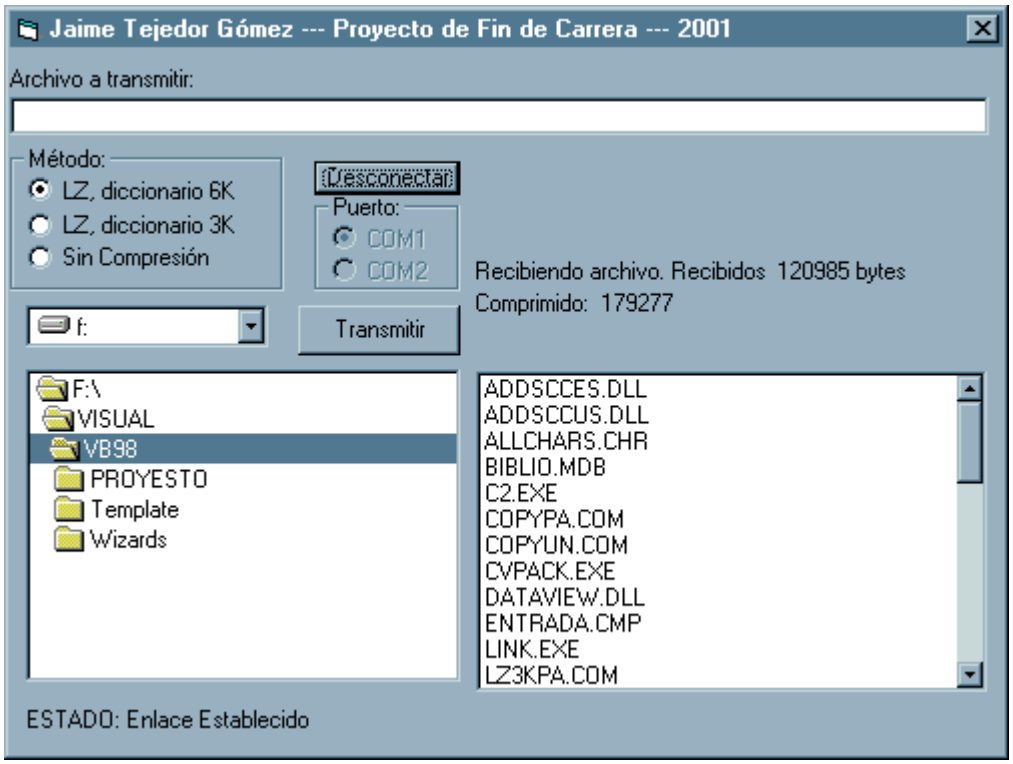
Captura 3.2.3.5: Comprimiendo



Captura 3.2.3.6: Enviando

Podemos ver como el archivo comprimido apenas ocupa un 6.4% del original, lo cual es una razón de compresión muy buena. Una vez comprimido el archivo, se pasa a

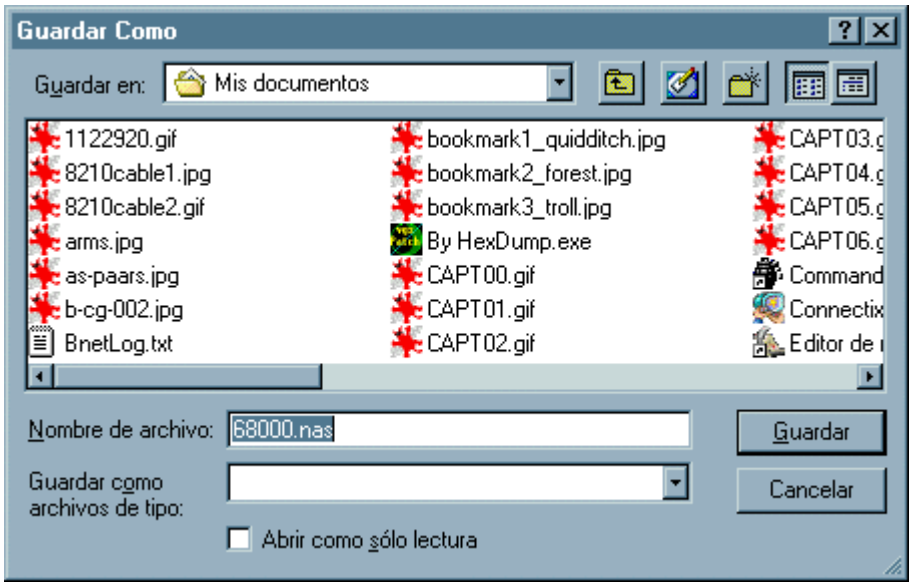
enviarse directamente por el puerto serie. Entonces vemos los cambios en el programa del lado receptor, como se puede observar en la captura 3.2.3.7:



Captura 3.2.3.7: Recibiendo

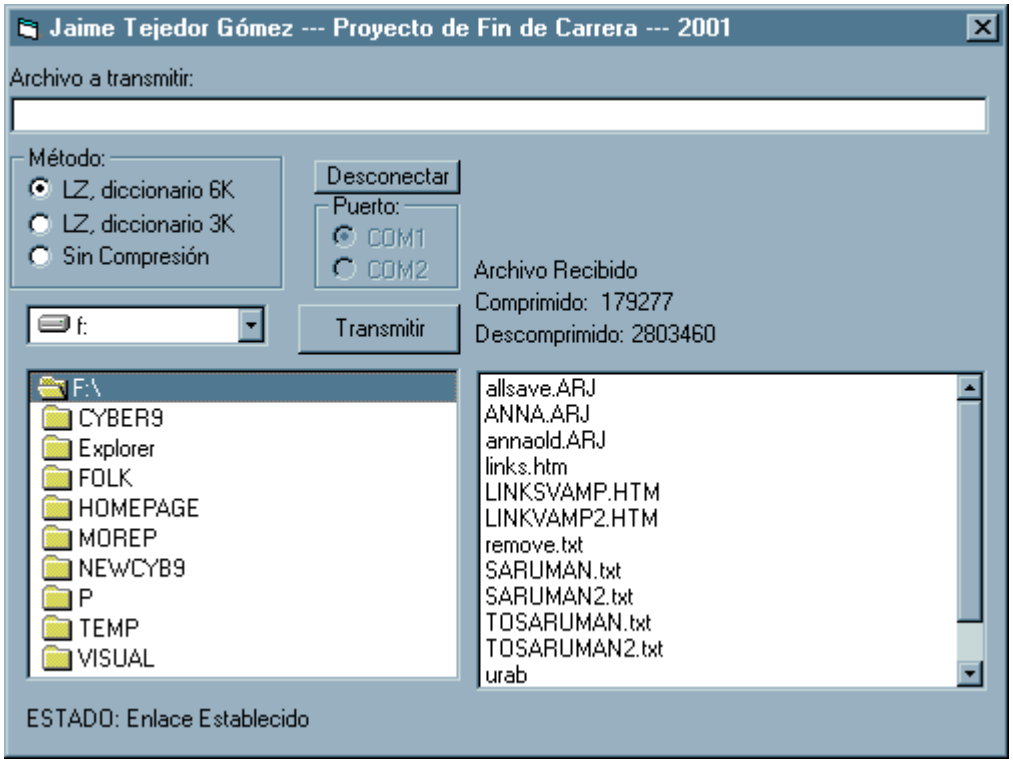
Antes de enviar el archivo comprimido byte a byte, lo primero que se envía es el tamaño que ocupa comprimido (de forma que el programa receptor sabe cuando debe dejar de escribir datos en el archivo de llegada), así como el nombre del archivo original (sin el path, por que no podemos esperar que si tenemos dos ordenadores diferentes, tengan el mismo árbol de directorios). De esta forma el programa receptor no solo informa cuantos bytes se han recibido hasta ahora sino también cuantos tienen que recibirse en total.

Una vez que llega el archivo comprimido, se abre una ventana nueva para escoger la localización del archivo a guardar, ya que los codecs de descompresión toman como entrada el archivo temporal donde se ha grabado el archivo comprimido que acabamos de recibir, y otro archivo donde se especifica el nombre y localización que tendrá el archivo a descomprimir, y por lo tanto necesitamos saber esa información antes de invocarlo. Por defecto se abre la carpeta “Mis Documentos”. Esto se ve en la captura 3.2.3.8.



Captura 3.2.3.8: Guardar archivo recibido

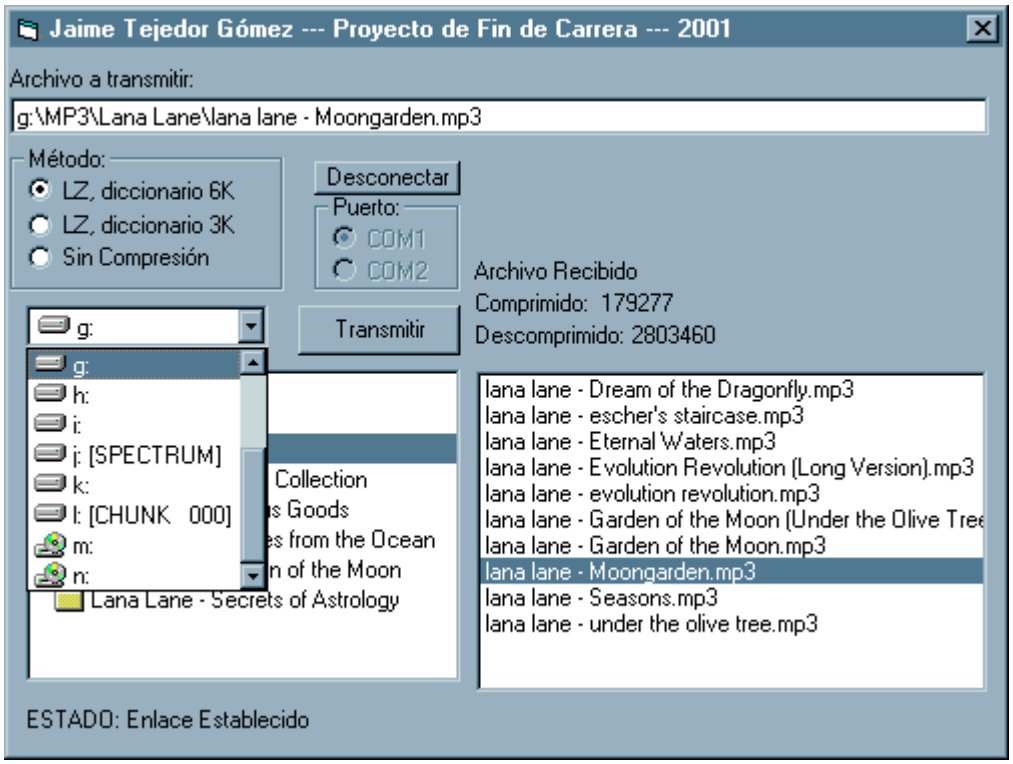
Una vez que escogemos el nombre (que por defecto es el que recibimos al comienzo de la transmisión), y la localización, pulsamos el botón de guardar. Al hacer esto, entra en acción el codec de descompresión. Una vez que ha acabado su trabajo, el programa principal nos informa sobre el tamaño original del archivo, tal y como puede verse en la siguiente captura 3.2.3.9:



Captura 3.2.3.9: Recepción finalizada

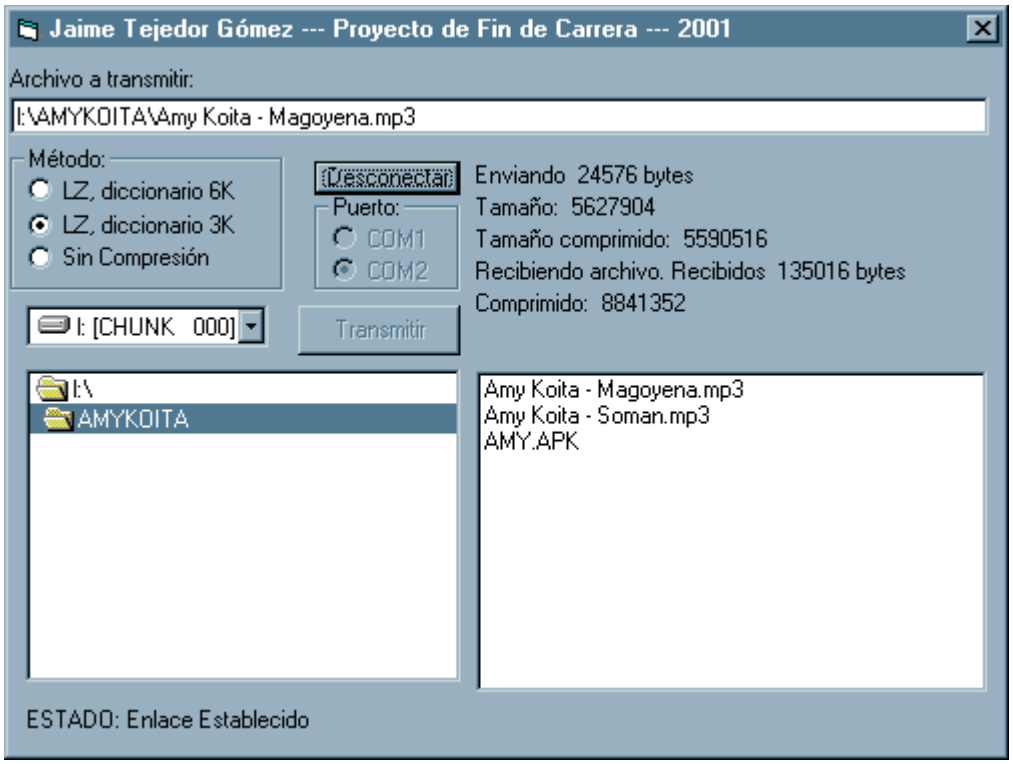
Vemos que el tamaño descomprimido coincide exactamente con el del archivo original, lo cual es una buena señal de que las cosas funcionan.

En las siguientes capturas veremos como funciona la transmisión full-duplex: Escogemos un archivo grande desde el programa que antes actuó de receptor. Es un MP3, por lo que hay muchas posibilidades de que no pueda comprimirse bien, dado que los archivos MP3 están muy comprimidos de por si (en la captura 3.2.3.10 se puede apreciar también un detalle de la selección de unidad).



Captura 3.2.3.10: Escogiendo unidad

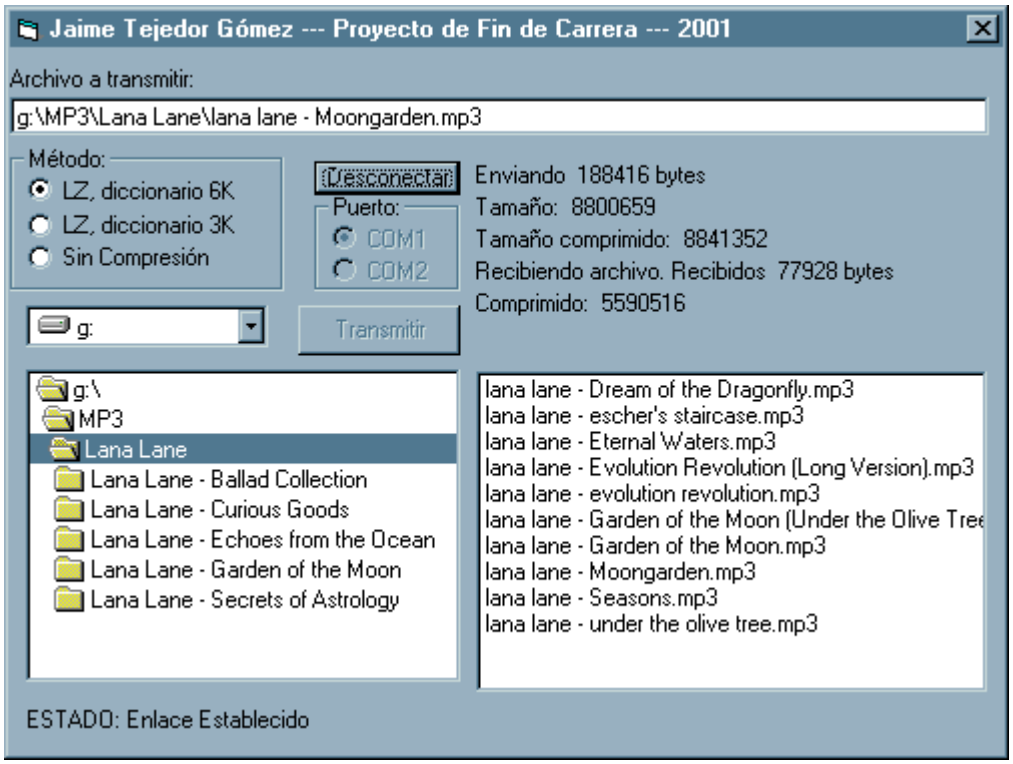
Una vez escogido el archivo, transmitimos. Ahora hacemos lo mismo desde el otro lado, en este caso cambiamos el método de compresión por el del diccionario de 3K. En la captura 3.2.3.11 podemos observar que a pesar de estar enviando otro MP3, en este caso si hemos obtenido compresión. Esto es debido a que los archivos MP3 pueden tener opcionalmente una o dos etiquetas de identificación, que son altamente redundantes. El primer archivo no tenía etiquetas y el segundo tenía dos. Por lo tanto en este caso no solo hemos logrado evitar la pérdida sino incluso obtener algo de ganancia.



Captura 3.2.3.11: Recibiendo un nuevo archivo se borra la línea de abajo

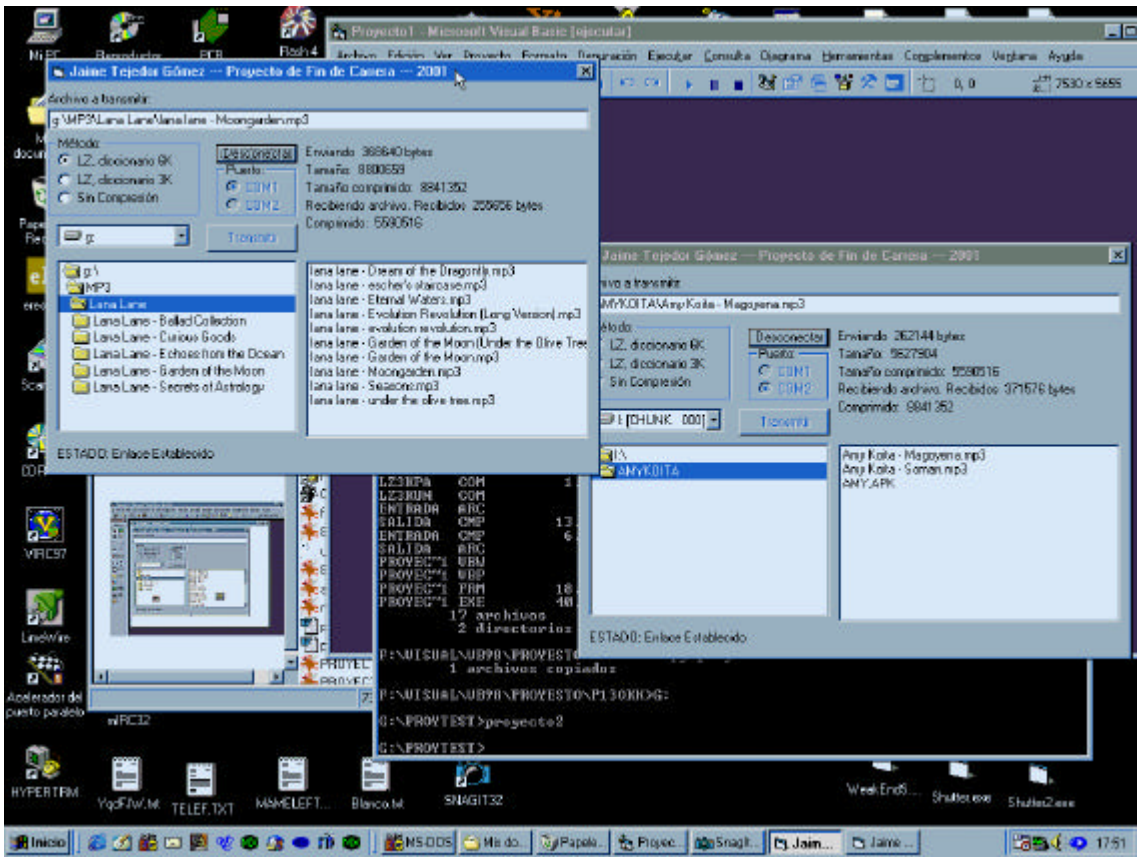
En la captura 3.2.3.12 podemos ver como la información del archivo recibido anterior se borra cuando estamos recibiendo un nuevo archivo, de forma que no haya confusión: la línea de abajo del todo que antes contenía la información del archivo que se envió previamente se ha borrado. También podemos ver la pérdida en el archivo MP3 de Lana Lane: Apenas se ha producido un incremento de tamaño de 40Kbytes sobre 8,8 Megas, lo cual indica que la técnica de las cadenas de literales funciona.

Otro detalle que puede observarse en las capturas 3.2.3.11 y 3.2.3.12 es que la comunicación es perfectamente full-duplex, ya que enviamos y transmitimos datos al mismo tiempo.



Captura 3.2.3.12

Finalmente, la captura 3.2.3.13 de la misma transmisión que muestra como ambos programas pueden funcionar perfectamente en el mismo ordenador al mismo tiempo:



Captura 3.2.3.13: Funcionamiento simultáneo en el mismo ordenador

3.2.3.3. Consideraciones sobre el código:

El código de Visual Basic es muy sencillo. Tan solo hay que codificar el evento automático que se produce al cargar el programa, el evento MSComm1_OnComm, que se encarga de gestionar todos los eventos relacionados con la comunicación y los elementos visuales que hemos colocado en el programa, como son la respuesta al click de los botones de opción, los botones de pulsación y los selectores de unidad, directorio y archivo.

El formulario de arranque de programa tan solo configura la opción inicial de método de compresión, el estado de escucha y almacena el directorio de trabajo actual para utilizarlo mas tarde.

Los botones de opción lo único que hacen es cambiar el valor de una variable, Packmethod en el caso de los botones correspondientes al método de compresión y en el caso de los correspondientes al puerto, se cambia directamente la propiedad del control MSComm1.

El cambio de directorio provoca sencillamente una actualización de lista archivos. El cambio de unidad, similarmente, provoca una actualización de la lista de directorios, pero en este caso se tiene en cuenta además que una unidad puede o no estar lista, y si no lo está provocará un error. Si cambiamos de unidad, cambiará también el directorio, por lo cual se activará automáticamente el evento de cambio de directorio aunque no hayamos pulsado con el ratón, y se actualizará la lista de archivos apropiadamente.

Si elegimos un archivo, su nombre y su path aparecen en el cuadro de texto, y se guarda su nombre suelto en la variable FileNameSend, de forma que queda listo para ser enviado. El nombre del archivo también puede escribirse directamente en el cuadro de texto.

Al pulsar el botón de conectar, intenta conectarse al puerto seleccionado. Si no lo consigue (por que el puerto ya está en uso), muestra una ventana de error y sale. Si se puede conectar, se cambia el texto (caption) del botón de forma que se convierte en Desconectar. Al mismo tiempo se deshabilitan los botones de opción de selección de

puerto (no se puede seleccionar puerto mientras está conectado), se pone en estado de espera de orden y se envía un código de Ping, y se cambia la etiqueta para que rece: "ESTADO: Conectado".

Al pulsar ese mismo botón cuando estamos conectados, sencillamente desconectamos el puerto, devolvemos el texto original al botón, habilitamos los botones de opción de selección de puerto y volvemos a colocar el texto "ESTADO: Sin conexión".

El código de Ping, al igual que el de Pong y los de transmisión de archivo han sido escogidos arbitrariamente, dado que solo necesitamos 5 códigos y podemos escoger entre 256 posibles.

Al pulsar el botón de transmitir, lo primero que se hace es comprobar si existe un archivo seleccionado correcto en el cuadro de texto, y si estamos conectados. Si alguna de estas condiciones no se cumplen aparecerá una ventana de error. Si no hay errores, deshabilitaremos el botón poniendo su propiedad Enable como False.

Para enviar un archivo, se salvan la unidad y directorio actuales (que corresponderán a los seleccionados en las listas), y pasamos al directorio de trabajo donde está el programa principal y los codecs de compresión y descompresión.

Ahora creamos en el directorio de trabajo el archivo ENTRADA.ARC, conteniendo el path completo del archivo que vamos a comprimir (o copiar, si empleamos el método "Sin compresión") y acabado con un byte puesto a 0 (esto último es necesario para el funcionamiento de las funciones de la API a las que acceden los codecs utilizando int 21h).

Ahora volvemos a restaurar las unidades que estaban seleccionadas, cambiamos el texto de las etiquetas de texto para que aparezcan los mensajes "Comprimiendo" y "Tamaño: " con el tamaño del archivo seleccionado.

El siguiente paso es invocar el codec de compresión, cuyo nombre depende del método de compresión seleccionado. Para ejecutar programas externos en modo MSDOS, tenemos que emplear la función shell, que se ejecuta de forma asíncrona, por lo que no

podemos esperar que una aplicación termine su ejecución antes de que se ejecuten las siguientes sentencias que siguen a la función. Para solucionar este problema (pues necesitamos que el codec termine de generar el archivo para poder enviarlo), usamos las funciones de la API de Windows GetActiveWindow(), GetForegroundWindow() e IsWindow(). Estas funciones son externas a Visual Basic, encontrandose en el archivo USER32.DLL, por lo que deben ser declaradas al comienzo del programa. La función GetActiveWindow() devuelve el handle correspondiente a la ventana activa asociada con el proceso que llama a la función, la función IsWindow() devuelve un valor True si la ventana especificada existe, y la función GetForegroundWindow() devuelve el Handle de la ventana con la que el usuario está trabajando actualmente. Por lo tanto lo que se hace es tomar la ventana actual con GetActiveWindow(), lanzar el proceso Shell, esperar a que deje de estar activa la ventana actual y tomar el handle de la nueva ventana activa con la función GetForegroundWindow(), que será nuestra ventana MSDOS. Ahora esperamos con la función IsWindow hasta que la ventana MSDOS desaparezca, momento en el cual podemos estar seguros de que nuestra aplicación ha terminado.

El codec de compresión entonces ha efectuado la compresión y podemos estar seguros de que se ha creado un archivo llamado “SALIDA.CMP” que podemos enviar. El archivo “ENTRADA.ARC” ha dejado de sernos útil, por lo que nos deshacemos de él borrandolo con la orden Kill. Al enviar lo primero que enviamos es una orden que indica que viene un archivo, así como su método de compresión. Seguidamente se envían 4 bytes con el tamaño (los bytes menos significativos se envían antes), y el nombre del archivo acabado en 0. Seguidamente, se envía todo el archivo de principio a fin, en bloques del tamaño del buffer de salida, y actualizando la etiqueta con el número de bytes enviados. Dado que el tamaño del buffer de salida es de 8K, el número que aparece en dicha etiqueta es siempre múltiplo de 8192.

Una vez acabado el envío, actualizamos por última vez la etiqueta de texto, haciendo que ponga “Archivo Enviado”, cerramos el archivo comprimido “SALIDA.CMP”, lo eliminamos y volvemos a activar el botón de transmitir, para permitir la transmisión de un nuevo archivo.

Por último, el evento MSComm1.EvComm se encarga de controlar todos los eventos que pueden aparecer en la comunicación, de modo que al principio de su código se hace un Select Case con la propiedad CommEvent, para dar un tratamiento personalizado a cada suceso.

El caso mas importante que vamos a tener en cuenta en esta rutina es el caso comEvReceive, que corresponde a la llegada de datos. El resto de casos se corresponden con errores o eventos que generan algún mensaje, o bien son ignorados para que no afecten a la comunicación. Al recibir datos lo primero que se mira es el estado actual de recepción. Si estamos en estado 1, estamos esperando una orden. La orden puede ser Ping, Pong o bien una de las tres ordenes de envío de archivos (hay una para cada método de compresión). Cualquier otro byte que llegue supone un error en la comunicación y es ignorado.

Si llega una orden Ping, responderemos con un Pong, y pondremos en la etiqueta de abajo el mensaje “Enlace Establecido”. Si nos llega un Pong, ponemos el mismo mensaje pero sin enviar nada más.

Si la orden que llega es de envío de archivo, escogemos el método de descompresión dependiendo de la orden y cambiamos el estado de la recepción a estado 2.

Los estados del 2 al 5 se encargan de leer byte a byte la longitud del archivo enviado y formar la variable que contiene dicha longitud. Una vez leída la longitud, se informa en la etiqueta pertinente sobre el tamaño del archivo comprimido que está llegando. El estado 6 va formando el nombre del archivo carácter a carácter hasta que encuentra un cero, momento el que se abre el archivo de salida “ENTRADA.CMP”, y pasamos al estado 0.

El estado 0 se encarga de leer los bloques de bytes y grabarlos en el archivo hasta que se haya terminado la transmisión. En este estado, los bytes en lugar de leerse de uno en uno se leen a la vez todos los que haya en el buffer, para esto basta con fijar la propiedad InputLen de MSComm1 a 0. En todos los demás estados, la propiedad tiene el valor a 1. Esta propiedad hace que el número de bytes leídos en cada ocasión no sea fijo, sino que puede ser cualquier número comprendido entre 1 y el tamaño del buffer de

entrada (16384). Una vez leído cada fragmento, se escribe en el archivo, se actualiza el número de bytes leídos y el número de bytes que quedan por leer y se escribe en su sitio el mensaje “Recibiendo archivo. Recibidos xxxx bytes”. Dado que la longitud de entrada no es fija, el número de bytes leídos se actualiza de forma mucho mas rápida y aleatoria que el de los bytes escritos, que se actualizaba exclusivamente cada 8192 bytes.

Una vez que la longitud restante cae a 0, cerramos el archivo y abrimos una la caja de dialogo estándar del tipo Guardar Como a la que damos el nombre que hemos recibido del archivo como nombre por defecto. Si el usuario pulsa cancelar en dicha ventana, borramos el archivo comprimido y no lo grabamos en ninguna parte. Si se escoge un nombre válido de archivo, creamos el archivo “SALIDA.ARC” para que el programa descompresor sepa donde tiene que guardar su salida. Ahora repetimos la operación que se hizo para comprimir el archivo para descomprimir el que ha llegado. Hacemos que en la etiqueta que informaba de los bytes recibidos aparezca “Descomprimiendo Archivo”, y llamamos al programa descompresor con la función shell de la misma forma que antes. Una vez descomprimido, informamos sobre el número de bytes que ocupa el archivo descomprimido, borramos los archivos “SALIDA.ARC” y “ENTRADA.CMP” y ya podemos pasar de nuevo al estado 1 con MSComm1.InputLen = 1 para procesar la siguiente orden y ser capaces de transmitir otro archivo.

3.2.4. PRUEBAS DE COMPRESIÓN

Resulta interesante conocer hasta que punto el grado de compresión alcanzado es bueno o no. Por lo tanto, vamos a escoger unos cuantos sets de archivos para comprimirlos tanto con nuestros codecs como con otros programas externos para compararlos.

Podemos anticipar que nuestros programas van a salir bastante mal parados, dado que hemos limitado deliberadamente la cantidad de memoria usada para adaptarla a las características técnicas de la placa hardware, y también hemos escogido un algoritmo sencillo sin ningún codificador de entropía. La mayoría de los compresores a los que nos vamos a enfrentar utilizan un diccionario de 64K o mayor y codificación de entropía Huffman o bien aritmética. Para equilibrar la situación, también he incluido algunos compresores mas sencillos con los que se puede competir con mayor igualdad de oportunidades.

Un detalle importante es que no es posible hacer una estimación correcta de los tiempos de compresión, dado que el caché de archivos de Windows tiende a falsear mucho los resultados, de modo que al comprimir un mismo archivo por segunda vez, la segunda es mucho más rápida porque el archivo estaba ya en el caché. Aparte de eso, el ordenador donde se hacen las pruebas es muy rápido (AMD Athlon K7 - 700MHz), por lo que la mayoría de las compresiones son casi instantáneas.

Es por esto que solo incluiremos los tamaños de compresión en las pruebas.

Los compresores a los que vamos a enfrentar a nuestros codecs particulares van a ser:

- LZSS: Compresor basado en LZ77, sin codificador de entropía.
- LZOP: Compresor sencillo diseñado para la compresión en tiempo real, buscando la velocidad ante todo. Basado en la librería LZO. Vamos a probar 3 variaciones de velocidad, la 1 es la mas veloz, y la –9 es la que mayor compresión logra, mientras que la 5 es un grado intermedio.
- APPACK: Programa de ejemplo de la librería aPLib.
- ARJ: Famoso programa de compresión muy extendido

- 7ZIP: Programa que crea archivos .ZIP de menor tamaño que otros enzipadores.
- RAR32: Famoso programa de compresión, muy extendido.
- BZIP2: Programa basado en la técnica BWT.
- RDMC: Programa basado en la técnica DMC.
- PPMD: Programa basado en la técnica PPM.

Los conjuntos de archivos que vamos a someter a prueba van a ser los 18 archivos del set Calgary Corpus, los 11 set Canterbury Corpus, un archivo MP3 con etiquetas ID1 e ID2, y un archivo previamente comprimido .ZIP

3.2.4.1. Calgary Corpus

Este es un clásico de los tests de compresión, junto al Canterbury Corpus. Se trata de 18 archivos de variada naturaleza, incluyendo textos, código objeto, gráficos, texto cogido de las news...

Los resultados de las pruebas pueden resumirse en las siguientes tablas:

ARCHIVO:	bib	Book1	Book2	Geo	news	Obj1	Obj2	Paper1
(sin comprimir)	111261	768771	610856	102400	377109	21504	246814	53161
PPMD	25521	215944	149543	55850	109657	9530	73224	14995
RDMC	30481	258552	180553	60833	138735	10787	86231	18141
BZIP2	27467	232598	157434	56291	118540	10906	76441	16558
RAR32	32985	281455	184214	66897	126575	9873	73624	18216
7ZIP	34014	300643	197787	65934	140301	10327	78631	17975
ARJ	36138	319158	210445	69094	146978	10432	82064	18762
APPACK	37102	345552	217887	70269	147509	9879	77610	19902
LZOP –9	39388	360589	235645	74291	162501	11222	88721	21002
LZOP –5	54892	477573	318698	92300	207765	12732	114343	27474
LZOP –1	54291	473069	316146	92301	205547	12760	114359	27351
LZSS	52591	424147	285942	83183	194435	12247	103002	24467
LZ6KPA	51628	435087	285032	81328	188582	11102	94582	24095
LZ3KPA	57413	463252	308981	82369	200922	11251	98728	26149

Tabla 3.2.4.1 – Calgary Corpus – parte 1

ARCHIVO:	Paper2	Paper3	Paper4	Paper5	Paper6	Pic	progc	progl	Progp	Trans
(sin comprimir)	82199	46526	13286	11954	38105	513216	39611	71646	49379	93695
PPMD	22892	14436	4624	4299	11106	50279	11362	14865	10255	17130
RDMC	26585	17025	5476	5089	13366	52539	13621	17782	12316	22463
BZIP2	25041	15837	5188	4837	12292	49759	12544	15579	10701	17899
RAR32	28733	17900	5567	5032	13158	50336	13218	15925	10869	18110
7ZIP	28441	17456	5449	4964	12973	50952	13023	15635	10872	18444
ARJ	30042	18237	5641	5103	13480	55199	13527	16529	11409	20191
APPACK	32325	19747	5975	5346	14198	56531	14240	17151	11679	19551
LZOP –9	33949	20790	6458	5853	15260	63943	15439	18798	12961	21664
LZOP –5	45378	27198	7974	7111	19543	86312	19660	26164	17170	29869
LZOP –1	44956	27026	7988	7125	19441	86327	19683	25924	17055	29591
LZSS	39703	23485	6798	6069	17609	105311	17531	22521	15445	33641
LZ6KPA	40192	23727	6776	5978	17096	66916	16646	20081	13645	27847
LZ3KPA	43260	25458	7114	6239	18589	67185	18319	22233	15916	34997

Tabla 3.2.4.2 – Calgary Corpus – parte 2

3.2.4.2. Canterbury Corpus

Este es un set parecido al Calgary Corpus, pero tiene 11 archivos de naturaleza mas variada tal vez.

Los resultados de las pruebas pueden resumirse en las siguientes tablas:

ARCHIVO:	Alice29.txt	Asyoulik.txt	Cp.html	Fields.c	Grammar.lsp	Kennedy.xls
(sin comprimir)	148481	125179	24603	11150	3721	1029744
PPMD	39915	36588	6718	2747	1080	124436
RDMC	45496	41385	8234	3354	1334	177075
BZIP2	43206	39569	7264	3039	1299	130280
RAR32	50640	46906	7981	3149	1295	125524
7ZIP	51281	46787	7831	3127	1286	176919
ARJ	54488	49645	8111	3235	1333	202693
APPACK	58425	53860	8633	3299	1351	211760
LZOP –9	60936	55987	9434	3724	1554	326295
LZOP –5	82374	73911	11696	4745	1834	360363
LZOP –1	81694	73243	11676	4713	1837	358098
LZSS	72406	65551	10941	3841	1537	288123
LZ6KPA	73461	66309	10253	3603	1430	343580
LZ3KPA	78833	70327	10984	3774	1432	262641

Tabla 3.2.4.3 : Canterbury Corpus - parte 1

ARCHIVO:	Lcet10.txt	Plrabn12.txt	Ptt5	Sum	Xargs.1
(sin comprimir)	419235	471162	513216	38240	4227
PPMD	101264	134701	50279	12005	1518
RDMC	122202	157507	52539	14443	1854
BZIP2	107626	145545	49759	12909	1762
RAR32	127027	175589	50336	11753	1801
7ZIP	136806	184476	50952	12307	1794
ARJ	145285	196999	55199	13122	1851
APPACK	152650	215579	56531	12037	1881
LZOP –9	163472	223124	63943	14264	2116
LZOP –5	221112	294969	86312	17628	2509
LZOP –1	219498	292313	86327	17742	2512
LZSS	197791	261943	105311	17803	2124
LZ6KPA	198535	269589	66916	16082	2053
LZ3KPA	216012	286927	67185	16836	2071

Tabla 3.2.4.4 : Canterbury Corpus - parte 2

3.2.4.3. Los archivos ya comprimidos

Vamos ahora a ver los resultados para los dos archivos que faltan:

ARCHIVO:	Amy Koita – Magoyena.mp3	HC14RESB.ZIP
(sin comprimir)	5627904	54097
PPMD	5554332	50586
RDMC	5734695	53287
BZIP2	5499686	51287
RAR32	5525667	50174
7ZIP	5547440	51062
ARJ	5535529	51147
APPACK	6016230	54335
LZOP –9	5560475	50392
LZOP –5	5562667	50970
LZOP –1	5562629	50829
LZSS	6201740	57326
LZ6KPA	5591160	51560
LZ3KPA	5590516	51799

Tabla 3.2.4.5: Archivos ya comprimidos

4. CONCLUSIONES

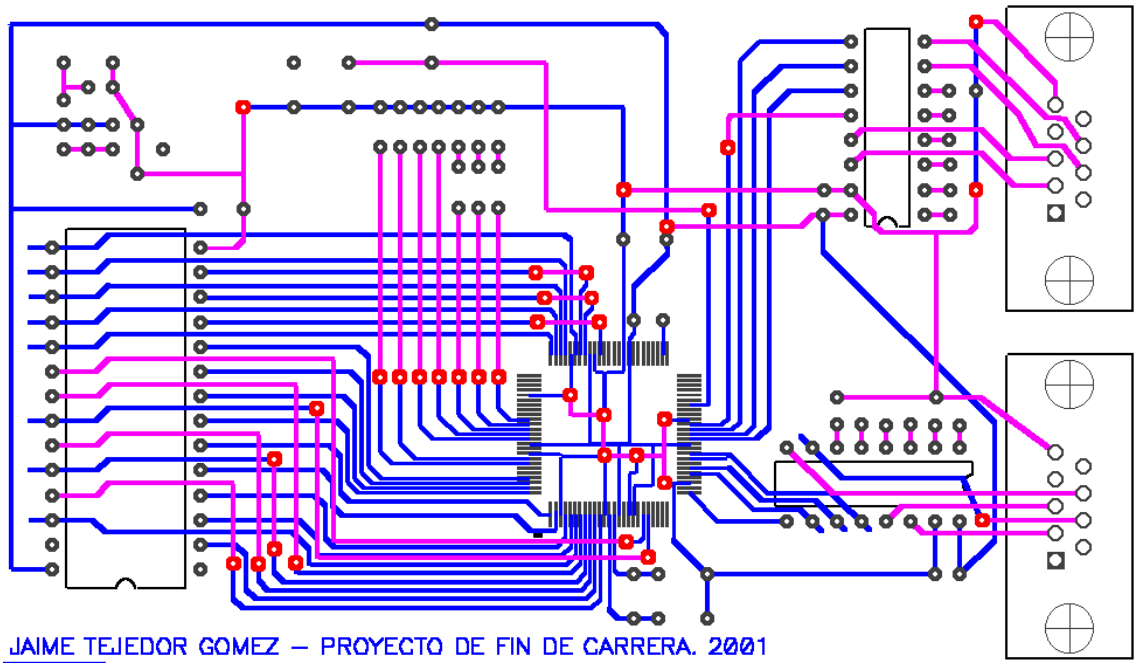
La placa hardware funciona y tiene un diseño sencillo a la vez que funcional sin ser demasiado costosa.

Los codecs de compresión han funcionado perfectamente demostrando su velocidad y resultados, ya que aunque no obtengan un grado de compresión espectacular, esto no era lo que se pretendía, y en la comparación con otros programas, a mis creaciones no les ha ido tan mal después de todo, ya que dentro de sus limitaciones han obtenido unos resultados bastante aceptables, logrando vencer en la mayor parte de las ocasiones al programa LZSS, y venciendo a su vez sobre LZOP cuando este usa la opción –5 o –1.

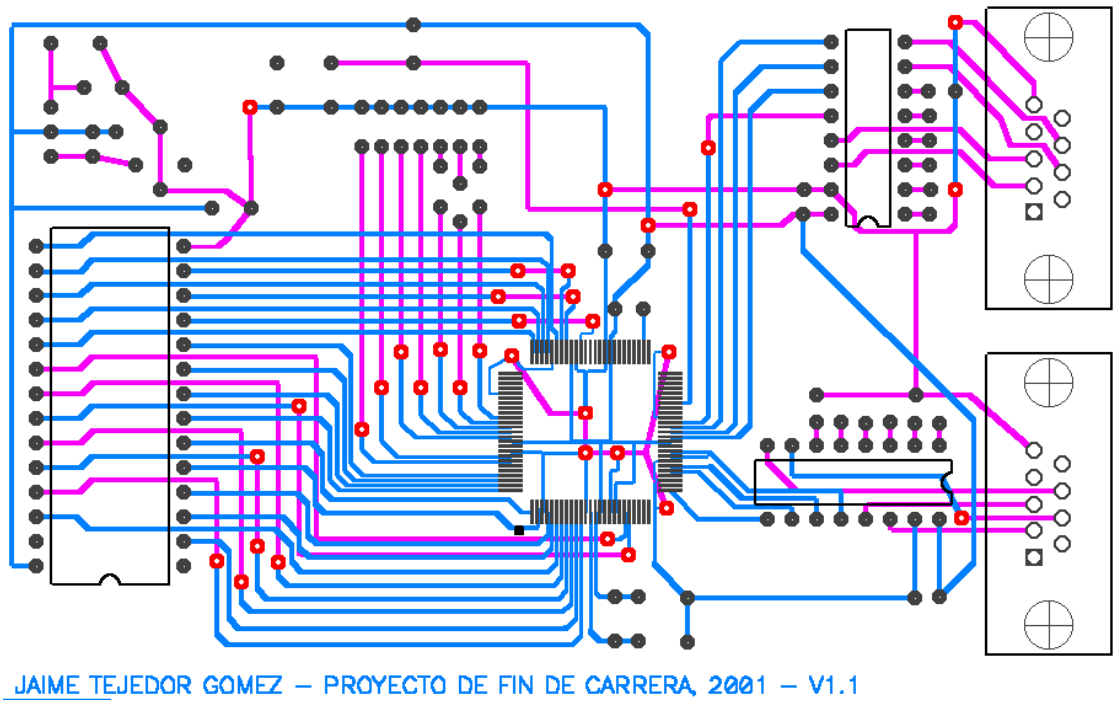
El algoritmo es lo suficientemente rápido como para poder ejecutarse en tiempo real y lo suficientemente efectivo como para ahorrar mucho tiempo de comunicación reduciendo la cantidad de datos que han de enviarse, mientras que al aplicarse sobre datos previamente comprimidos apenas aumenta el tamaño.

Por esto considero que los objetivos del proyecto se han logrado satisfactoriamente.

5. PLANOS



Plano 5.1: PCB de la placa. La capa Top está en azul, la Bottom en magenta, las conexiones en negro y las vías en rojo.



Plano 5.2: Rutado previsto para la segunda placa (no realizada).

6. PRESUPUESTO

El presupuesto estimado del proyecto tiene dos partes bien diferenciadas: La parte Hardware y la parte Software. Para la parte Hardware, necesitaríamos 2 placas (en caso, de haberse realizado completamente) y por eso hay el doble de cada elemento:

ADSP-2186L	2	3000	6000
MAX3232	4	550	2200
EPROM M27C64A	2	700	1400
DB9M	4	125	500
TIRAS DE PINES	4	100	400
XTAL 16MHz	2	100	200
C 22nF	4	10	40
C 0.1μF	26	15	390
C 1μF	2	15	30
LED AMARILLO	2	10	20
LED ROJO	2	10	20
LED VERDE	2	10	20
R 220 Ω	2	5	10
R 10K Ω	12	5	60
R 330 Ω	6	5	30
POTENCIOMETRO 500Ω	2	20	40
LM 317	2	75	150
PULSADOR	2	50	100
CABLE NULL-MODEM	3	900	2700

Por su parte, NASM (el ensamblador usado para los codecs) es gratuito, por lo que el único artículo del presupuesto para Software es:

VISUAL BASIC V6.0 ED. PROFESIONAL LICENCIA PARA ESTUDIANTES	1	25475	25475

Presupuesto total: 39785

7. FUENTES DE INFORMACIÓN

7.1 WEBSITES:

http://www.analog.com

http://www.motorola.com

http://www.ti.com

http://www.necel.com

http://www.hitachi.com

http://www.zilog.com

http://www.arturocampos.com

http://act.by.net

http://artest1.tripod.com

7.2 BIBLIOGRAFÍA:

Serial Port Complete, de Jan Axelson. 1998 Lakeview Research

Enciclopedia de Microsoft Visual Basic 4, de Fco. Javier Ceballos Sierra. 1996 Ra-Ma

8. APÉNDICE: CÓDIGOS FUENTE

8.1. CODECS DE COMPRESIÓN

8.1.1. COPYPA.NAS

```
;COPYPA: Packer para el proyecto de fin de carrera
; (Método 3 - COPY > sin compresión)
;
; Realizado en lenguaje ensamblador, se compila con NASM
;
; - El programa maestro de Windows, para transmitir permite seleccionar
; un archivo visualmente, entonces graba el nombre del archivo con su unidad
; y path delante y acabado en 0 en otro archivo llamado ENTRADA.ARC, que se
; coloca en el mismo path del programa, donde también está situado este módulo
; de compresión.
;
; - Este modulo de compresión tiene como misión abrir el archivo mencionado
; en ENTRADA.ARC y comprimirlo en el archivo de salida SALIDA.CMP, que queda
; listo para ser enviado por el puerto serie

                org     256             ;Programa

                mov     ah,3ch          ;Función para crear archivo
                xor     cx,cx           ;Atributos a 0
                mov     dx,nameout      ;Nombre del archivo: SALIDA.CMP
                int     33              ;Crear archivo de salida

                jnc     noerror01       ;Si no hay error, continuar
                mov     ax,4c01h        ;Código de error 1
                int     33              ;Salir a Windows
noerror01
                mov     [handleout],ax  ;Salvar handle de SALIDA.CMP

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namepath     ;Nombre del archivo a abrir
                int     33              ;Abrir ENTRADA.ARC

                jnc     noerror02       ;Si no hay error, continuar
                mov     ax,4c02h        ;Código de error 2
                int     33              ;Salir a Windows
noerror02
                xchg    ax,bx           ;Handle de ENTRADA.ARC en bx
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,400          ;Número máximo de bytes a leer
                mov     dx,pathbuffer   ;Posición donde se escriben los bytes
                int     33              ;Leer bytes

                jnc     noerror03       ;Si no hay error, continuar
                mov     ax,4c03h        ;Código de error 3
                int     33              ;Salir a Windows
noerror03
                mov     si,dx           ;Apuntar al nombre de archivo con path
                mov     dx,1            ;Abrir archivo existente, no crear
                xor     bx,bx           ;Modo de acceso solo lectura
                xor     cx,cx           ;Sin atributos
                mov     ax,716ch        ;Función para crear o abrir archivo
                                        ; usando nombres largos
                int     33              ;Abrir archivo de entrada

                jnc     noerror05       ;Si no hay error, continuar
```

```
                mov     ax,4c05h        ;Código de error 5
                int     33              ;Salir a Windows

noerror05       mov     [handlein],ax   ;Salvar handle del archivo de entrada

buclecopy       mov     ah,3fh
                mov     bx,[handlein]
                mov     cx,6000
                mov     dx,buffer
                int     33              ;Leer archivo

                jnc     noerror06
                mov     ax,4c06h        ;Código de error 6
                int     33              ;Salir a Windows
noerror06

                xchg    ax,cx           ;Número de bytes leídos en cx
                jcxz    exit            ;Si es 0, salir

                mov     ah,40h
                mov     bx,[handleout]
                int     33              ;Escribir archivo

                jnc     noerror07
                mov     ax,4c07h        ;Código de error 7
                int     33              ;Salir a Windows
noerror07
                jmp     buclecopy       ;Seguir copiando

exit            mov     ax,4c00h        ;Código de error 0 (sin problemas)
                int     33              ;Salir a Windows

namepath        db      "ENTRADA.ARC",0
nameout         db      "SALIDA.CMP",0

                                        ;*******************************
section .bss                            ;Variables sin iniciar


handleout       resw    1       ;Handle del archivo de salida
handlein        resw    1       ;Handle del archivo de entrada


pathbuffer      resb    400

buffer          resb    6000

;errorlevels:
;
; 0 - OK, no error
; 1 - No se puede crear SALIDA.CMP
; 2 - No se puede abrir ENTRADA.ARC
; 3 - No se puede leer ENTRADA.ARC
; 5 - No se puede abrir el archivo de entrada
; 6 –
```

8.1.2. LZ6KPA.NAS

```
;LZ6KPA: Packer para el proyecto de fin de carrera
; (Método 1 - LZ6K > LZ greedy con diccionario de 6K)
;
; Versión 1.2
;
; Realizado en lenguaje ensamblador, se compila con NASM
;
; - El programa maestro de Windows, para transmitir permite seleccionar
; un archivo visualmente, entonces graba el nombre del archivo con su unidad
```

```
; y path delante y acabado en 0 en otro archivo llamado ENTRADA.ARC, que se
; coloca en el mismo path del programa, donde también está situado este módulo
; de compresión.
;
; - Este modulo de compresión tiene como misión abrir el archivo mencionado
; en ENTRADA.ARC y comprimirlo en el archivo de salida SALIDA.CMP, que queda
; listo para ser enviado por el puerto serie

                                        ;*****************************
                org     256             ;Programa

start           mov     ah,3ch          ;Función para crear archivo
                xor     cx,cx           ;Atributos a 0
                mov     dx,nameout      ;Nombre del archivo: SALIDA.CMP
                int     33              ;Crear archivo de salida

                jnc     noerror01       ;Si no hay error, continuar
                mov     ax,4c01h        ;Código de error 1
                int     33              ;Salir a Windows
noerror01
                mov     [handleout],ax  ;Salvar handle de SALIDA.CMP

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namepath     ;Nombre del archivo a abrir
                int     33              ;Abrir ENTRADA.ARC

                jnc     noerror02       ;Si no hay error, continuar
                mov     ax,4c02h        ;Código de error 2
                int     33              ;Salir a Windows
noerror02
                xchg    ax,bx           ;Handle de ENTRADA.ARC en bx
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,400          ;Número máximo de bytes a leer
                mov     dx,pathbuffer   ;Posición donde se escriben los bytes
                int     33              ;Leer bytes

                jnc     noerror03       ;Si no hay error, continuar
                mov     ax,4c03h        ;Código de error 3
                int     33              ;Salir a Windows
noerror03
                mov     si,dx           ;Apuntar al nombre de archivo
                mov     dx,1            ;Abrir archivo existente, no crear
                xor     bx,bx           ;Modo de acceso solo lectura
                xor     cx,cx           ;Sin atributos
                mov     ax,716ch        ;Función para crear o abrir archivo
                                        ; usando nombres largos
                int     33              ;Abrir archivo

                jnc     noerror05       ;Si no hay error, continuar
                mov     ax,4c05h        ;Código de error 5
                int     33              ;Salir a Windows

noerror05       mov     [handlein],ax   ;Salvar handle del archivo de entrada

                xchg    ax,bx           ;Handle en bx
                mov     ah,3fh          ;Función para leer
                mov     cx,6144         ;6K van a ser leídos
                mov     dx,buffer       ;En el buffer
                int     33              ;Leer
                call    copytoend       ;Copiar los primeros 128bytes al final
                add     dx,ax           ;End of read bytes
                cmp     ax,cx           ;Si se leen menos de 6K
                jnc     nosmallfile
                mov     [EOF],dx        ;Marcar End of File
                jmp     short bucle     ;Y no poner readedge
nosmallfile     add     ax,buffer-128   ;Readedge > una vez procesado hasta
                mov     [readedge],ax   ; aquí, tenemos que leer 128 nuevos
                                        ; bytes
                ;La compresión comienza aquí
```

```
bucle           inc     word [maxdistance] ;Aumentar la distancia máxima de
                                           ;  búsqueda de coincidencia en 1 byte
                mov     si,[current]    ;Posición del byte actual
                cmp     si,[EOF]        ;Verificar si hemos llegado al final
                jnz     noEOFfound      ; del archivo, si no es así saltar
                mov     ax,[untagged_literals]
                and     ax,ax           ;Comprobar si quedan literales
                jz      nomoreflush     ; pendientes de marcar
                call    tagliterals     ;Marcarlos
nomoreflush     call    puttag1         ;Poner
                call    puttag0         ; código
                call    puttag0         ; de
                call    puttag0         ; fin
                call    puttag0         ; de
                call    puttag1         ; archivo
                call    puttag1         ; correspondiente
                call    puttag1         ; a 10000111
                call    puttag0         ;Rellenar con ceros para que se
                call    puttag0         ; escriba el último tag-byte y se
                call    puttag0         ; vacíen los literales que haya
                call    puttag0         ; pendientes
                call    puttag0         ;con 7 nos aseguramos de ello
                call    puttag0
                call    puttag0

                mov     ah,40h          ;Función de escritura
                mov     bx,[handleout]  ;Handle del archivo de salida
                mov     cx,[currentoutput] ;Puntero de escritura
                mov     dx,outbuffer    ;Dirección del buffer de escritura
                sub     cx,dx           ;Número de bytes a escribir
                int     33              ;Vaciar el buffer de escritura

end             mov     ax,4c00h        ;Sin error
                int     33              ;Salir a Windows


noEOFfound      push    si              ;Almacenar posición actual en la pila
                lodsb                   ;Leer byte
                cmp     si,endbuffer    ;Si la siguiente posición llega al
                jnz     nowrap00        ; final del buffer
                mov     si,buffer       ;colocarla al principio
nowrap00        mov     [current],si    ;actualizar posición de byte procesado
                cmp     si,[readedge]   ;si hemos llegado al límite de lectura
                jnz     noedgecrossed   ;hay que leer un nuevo bloque, si no
                                        ;se continúa

                pusha                   ;Guardar registros
                mov     cx,128          ;Bytes a leer / avanzar
                mov     dx,si           ;Se va a leer en
                add     dx,cx           ; la posición actual + 128
                cmp     dx,endbuffer    ;¿esto está al final del buffer?
                jnz     nowrap02        ;NO: saltar
                mov     dx,buffer       ;SI: colocar al comienzo
nowrap02        mov     ah,3fh          ;Función de lectura
                mov     bx,[handlein]   ;Handle del archivo de entrada
                int     33              ;Leer 128 bytes
                jnc     noerror04       ;Si no hay error, continuar
                mov     ax,4c04h        ;Código de error 4
                int     33              ;Salir a Windows
noerror04
                cmp     dx,buffer       ;Si hemos leido en el principio
                jnz     nocopytoend
                call    copytoend       ;Pasarlo al final también
nocopytoend
                cmp     ax,cx           ;Verificar si se han leido 128 bytes
                jz      EOFnotFOUND     ;Si es así saltar
EOFFOUND        add     dx,ax           ;Si se han leido menos, colocar
                mov     [EOF],dx        ; la marca de fin de archivo
                jmp     short NoReadEdge; y no fijar un nuevo readedge
EOFnotFOUND     mov     [readedge],dx   ;Actualizar readedge
```

```
NoReadEdge      sub     word [maxdistance],128 ;Actualizar maxdistance
                popa                    ;Restaurar registros
noedgecrossed   pop     si              ;Byte en al, si=current

                mov     bl,al
                mov     bh,0
                add     bx,bx           ;BX=AL*2

                mov     bp,[bx+prev]    ;Buscar última posición del byte en
                                        ; el buffer
                mov     [bx+prev],si    ;Actualizar última posición

                mov     cx,si           ;CX = distancia al previo, para
                sub     cx,bp           ; poner en prevbuff

                ja      nowrap03        ;Si la distancia es 0 o negativa
                add     cx,6144         ;Ajustarla
nowrap03                                ;CX=distancia entre si y bp
                mov     bx,si
                mov     [bx+si],cx      ;Actualizar prevbuff

                dec     byte[codepaired] ;Si el byte ya está comprimido en un
                jns     near bucle       ; codepair, no hace falta buscar
                inc     byte[codepaired] ; coincidencias

                and     bp,bp           ;Si bp es 0, el byte es un literal
                jz      near SETLITERAL_clearprevbuff ; colocarlo

                cmp     al,[bp]         ;Si el previo no coincide, era falso
                jnz     near SETLITERAL_clearprevbuff ;Por lo tanto > literal

                mov     byte  [max_match_lenght],0  ;Iniciar  distancia  máxima
hallada
                                        ; hasta el momento

        ;Entrada al bucle de búsqueda > Distancia acumulada en CX
        ;                             > Posición del siguiente a compara en BP
loop00          cmp     [maxdistance],cx ; si la distancia sobrepasa el límite
                jc      exitsearchloop  ;dejar de buscar

                push    si              ;Guardar posición actual

                mov     di,bp   ;SI=position_temp_curr
                                ;DI=position_temp_comp
                mov     dl,0    ;DL=Máxima longitud de coincidencia actual
keepmatching    cmpsb           ;Comparar bytes
                jnz     nomore  ;Si no coinciden, salir de esta comparación
                inc     dl      ;Si coinciden, incrementar número de bytes
                js      nomore  ;Si es 128, no seguir buscando
                cmp     si,[EOF];Si hemos llegado al final del archivo,
                jnz     keepmatching ; tampoco, en caso contrario seguir
nomore          cmp     [max_match_lenght],dl ;Comparar coincidencia con la
                                        ; máxima hasta ahora
                jnc     trynext         ;Si no la supera, pasar a la
                                        ; siguiente
                mov     [max_match_lenght],dl ;Si la supera, guardar la
                                        ; longitud
                mov     [max_match_distance],cx ; y la distancia del match
                cmp     dl,128          ;si es de longitud máxima
                jz      exitsearchmax   ; dejar de buscar

trynext         mov     di,bp           ;Para acceder a prevbuff[bp]
                add     cx,[bp+di]      ;Acumular distancia

                sub     bp,[bp+di]      ;Hallar nueva posición
                cmp     bp,buffer       ;Ver se pasa del buffer
                jnc     nowrap07        ;Si no, saltar
                add     bp,6144         ; Si se pasa, ajustar
nowrap07        pop     si              ;Recuperar posición actual
                jmp     short loop00    ;Repetir el bucle de búsqueda
```

```
exitsearchmax   pop     si              ;Limpiar pila
exitsearchloop  cmp     [max_match_lenght],byte 2  ;Si la distancia es menor
                jc      near SETLITERAL ; a 2, poner un literal
                jnz     setcodepair     ; si es mayor, poner un codepair
                cmp     [max_match_distance],word 255 ;Si es = 2, depende de
                jnc     SETLITERAL      ; la distancia. Mayor a 255 > literal

setcodepair       mov       cx,[untagged_literals] ;Ver si hay literales sin
marcar
                jcxz    notagliterals   ;Si no, saltar
                call    tagliterals     ;Marcar los literales pendientes
notagliterals   mov     al,[max_match_lenght] ;Fijar el número de
                dec     ax              ; bytes que no deben
                mov     [codepaired],al ; realizar la búsqueda

                call    puttag1         ;Codificar un 1
                mov     bx,[max_match_distance] ; BX=distancia
                bsr     cx,bx           ;Número de bits que van a ser
                                        ; codificados
                btr     bx,cx           ;Eliminar el bit mayor
                bt      cx,3            ;Colocar el bit 3 del número de bits
                call    puttagbit       ; de la distancia
                bt      cx,2            ;Excitar flag C según el bit 2
                call    puttagbit       ;Codificar bit
                bt      cx,1            ;Ahora el 1
                call    puttagbit       ;Codificar bit
                bt      cx,0            ;Y por último el 0
                call    puttagbit       ;Codificar bit
                jcxz    x_xdone         ;Si no hay que poner mas bits, la
                                        ; distancia ya está hecha
                shl     bx,1            ;Adaptar bx
putx_x          bt      bx,cx           ;Averiguar bit de la distancia
                call    puttagbit       ;Codificarlo
                loop    putx_x          ;Colocarlos todos
x_xdone         mov     bl,[max_match_lenght] ;
                xor     bh,bh           ;BX = longitud de la coincidencia
                dec     bx              ;Codificar uno menos, ya que el minimo
                                        ; es 2
                bsr     cx,bx           ;Número de bits a codificar
                btr     bx,cx           ;Eliminar el bit mayor
                bt      cx,2            ;Colocar los 3 bits del número de bits
                call    puttagbit       ; de la longitud
                bt      cx,1            ;Al igual que se hizo antes con los
                call    puttagbit       ; de la distancia
                bt      cx,0            ;Último bit...
                call    puttagbit       ;Codificarlo
                jcxz    y_ydone         ;Saltar si no hay bits que colocar
                shl     bx,1            ;Adaptar bx
puty_y          bt      bx,cx           ;Excitar flag C
                call    puttagbit       ;Codificar bit
                loop    puty_y          ;Hacerlo con todos
y_ydone         jmp     bucle           ;Volver al bucle principal

SETLITERAL_clearprevbuff
                mov     [bx+si],word 6143 ;Colocar en prevbuff una distancia
                                        ; tal que se pasa y no sigue buscando
SETLITERAL      mov     bx,[untagged_literals] ; Número de literales sin
                                        ; marcar
                and     bx,bx           ;Comprobar si es el primero
                jnz     nofirstuntagged ;Si no lo es saltar
                mov     [untagged_literal_start],si ; Si es el primero,
                                        ; colocar dirección de comienzo de
                                        ; los literales
nofirstuntagged inc     bx              ;Incrementar número de literales
                mov     [untagged_literals],bx ;Guardarlo
                cmp     bx,259          ;Comprobar si se ha llegado al máximo
                jnz     y_ydone         ;Si no, volver al bucle principal
                push    word bucle      ;Si se llega al máximo, ejecutar
                                        ; tagliterals y volver al bucle
```

```
tagliterals     pusha                   ;Guardar registros
                mov       bx,[untagged_literals]  ;  Número  de  literales  sin
marcar
                mov     bp,bx           ;También en BP
                cmp     bx,8            ;Si es menor que 8
                jc      plain_literals  ;Ir a plain_literals
                call    puttag1         ;Si no
                call    puttag1         ;Colocar los 3 bits que indican
                call    puttag1         ; cadena de literales
                sub     bx,4            ;Restar 4 al número de bytes
                bsr     cx,bx           ; que va a ser codificado
                btr     bx,cx           ;Eliminar el bit mayor
                bt      cx,2
                call    puttagbit       ; Colocar el número de bits
                bt      cx,1            ;del número de bits de la cadena
                call    puttagbit
                bt      cx,0
                call    puttagbit
                dec     cx              ;Decrementar cx
puty            bt      bx,cx           ;Coger bit
                call    puttagbit       ;Codificarlo
                loop    puty            ;Repetir con todos

                mov     ax,[tagged_literals]    ;Ver si hay literales ya
                and     ax,ax           ; marcados
                jnz     noanotherstart  ;Si los hay, no cambiar el comienzo
                mov     dx,[untagged_literal_start] ;Coger el comienzo de los
                                        ; no marcados
                mov     [tagged_literal_start],dx ; Y colocarlo en el comienzo
                                        ; de los marcados
noanotherstart   add      [tagged_literals],bp     ;Incrementar número de bytes
marcados
                xor     ax,ax                   ;AX=0
                bt      bx,ax                   ;Último bit por colocar
                call    puttagbit               ;Codificarlo
                mov     [untagged_literals],ax  ;Literales sin marcar=0
                popa                            ;Restaurar registros
                ret                             ;Volver

plain_literals  mov     ax,[tagged_literals]    ;Ver si ya hay literales
                and     ax,ax           ; marcados
                jnz     noanotherstart2 ;Si los hay, no cambiar el comienzo
                mov     dx,[untagged_literal_start] ;No hay: Poner el comienzo
                                        ; de los no marcados
                mov     [tagged_literal_start],dx ; en el de los marcados
noanotherstart2 inc     word [tagged_literals]  ;Incrementar número de
                                        ; literales marcados
                call    puttag0                 ;Codificar un 0
                inc     word [untagged_literal_start] ;Incrementar comienzo de
                                        ; literales sin marcar
                dec     word [untagged_literals];Decrementar número de
                                        ; literales sin marcar
                jnz     plain_literals  ;Repetir mientras queden
                popa                    ;Restaurar registros
                ret                     ;Volver

copytoend       pusha                   ;Guardar registros
                mov     si,buffer       ;Origen: comienzo del buffer
                mov     di,endbuffer    ;Destino: final del buffer
                mov     cx,64           ;64 words
                rep     movsw           ;Copiar
                popa                    ;Restaurar registros
                ret                     ;Volver

puttag0         clc                     ;Flag C = 0
                jmp     short puttagbit ;Ir a poner el bit
puttag1         stc                     ;Flag C = 1
puttagbit       pusha                   ;Guardar registros
                mov     dl,[tag_bits]   ;Coger tag_bits actuales
```

```
                adc     dl,dl           ;Colocar nuevo bit
                jnc     nonexttagbyte   ;Si no sale el marker bit, saltar
                mov     al,dl           ;Si sale, colocar el al
                call    writebyte       ;Y escribir el byte
                mov     cx,[tagged_literals] ;Comprobar si hay literales
                                             ; marcados
                jcxz    notaggedliterals ; Si no, saltar
                mov     si,[tagged_literal_start] ;Si los hay, poner el origen
                                                  ; en si
putliterals     lodsb                   ;Cargar literal marcado
                call    writebyte       ;Escribirlo
                cmp     si,endbuffer    ;Si llegamos al final del buffer
                jnz     nowrap08
                mov     si,buffer       ;Colocar al principio
nowrap08        loop    putliterals     ;Hacer todos los literales marcados
                mov     [tagged_literals],cx ;Ahora literales marcados=0
notaggedliterals mov     dl,1           ;Nuevo marker-bit
nonexttagbyte   mov     [tag_bits],dl   ;Actualizar tag_bits
                popa                    ;Restaurar registros
                ret                     ;Volver

writebyte       pusha                   ;Guardar registros
                mov     di,[currentoutput] ;Posición actual del puntero de
                                           ; escritura
                stosb                   ;Escribir el byte en memoria

                cmp     di,endoutbuff   ;¿El puntero llega al final?
                jnz     nolastbyte      ;NO: Saltar
                mov     ah,40h          ;Función de escritura
                mov     bx,[handleout]  ;Handle del archivo de salida
                mov     cx,32768        ;Número de bytes
                mov     dx,outbuffer    ;Buffer de escritura
                int     33              ;Escribir bytes

                jnc     noerror07       ;Si no hay error, continuar
                mov     ax,4c07h        ;Código de error 7
                int     33              ;Salir a Windows
noerror07
                mov     di,dx           ;Colocar puntero al principio
nolastbyte      mov     [currentoutput],di  ; Actualizar puntero de escritura
                popa                    ;Restaurar registros
                ret                     ;Volver

;FIN DEL PROGRAMA

                                        ;*****************************
                                        ;Variables con valores iniciados

EOF             dw      65535           ;(Basta con que esté fuera del buffer)


current         dw      buffer  ;Posición de memoria siendo procesada

namepath        db      "ENTRADA.ARC",0
nameout         db      "SALIDA.CMP",0

tagged_literal_start    dw    0       ;Comienzo de los literales marcados
tagged_literals         dw    0       ;Número de literales marcados
untagged_literal_start  dw    0       ;Comienzo de los literales sin marcar
untagged_literals       dw    0       ;Número de literales sin marcar


tag_bits        db      1               ;Tag byte en construcción

codepaired      db      0               ;Número de bytes a saltar

currentoutput   dw      outbuffer
maxdistance     dw      -1      ;Hasta cuanto podemos retroceder al buscar
                                ; un match
readedge        dw      -1      ;Límite de proceso hasta volver a leer un
```

```
                                ; bloque de 128 bytes


prev            times   256 dw 0


                                        ;*******************************
                                        ;*******************************
section .bss                            ;Variables sin iniciar


handleout       resw    1       ;Handle del archivo de salida
handlein        resw    1       ;Handle del archivo de entrada

max_match_lenght  resb    1     ;Máxima lóngitud de match hallado
max_match_distance resw    1    ;Distancia a la que se encuentra dicho match


pathbuffer      resb    400     ;Aquí se almacena la posición del último
                                ; byte de un determinado tipo

buffer          EQU     8192
endbuffer       EQU     buffer+6144
prevbuff        EQU     16384
outbuffer       EQU     28672
endoutbuff      EQU     outbuffer+32768

;errorlevels:
;
; 0 - OK, no error
; 1 - No se puede crear SALIDA.CMP
; 2 - No se puede abrir ENTRADA.ARC
; 3 - No se puede leer ENTRADA.ARC
; 4 - Error de lectura
; 5 - No se puede abrir el archivo de entrada
; 7 - Error de escritura
```

8.1.3. LZ3KPA.NAS

```
;LZ3KPA: Packer para el proyecto de fin de carrera
; (Método 1 - LZ3K > LZ greedy con diccionario de 3K)
;
; Versión 1.2
;
; Realizado en lenguaje ensamblador, se compila con NASM
;
; - El programa maestro de Windows, para transmitir permite seleccionar
; un archivo visualmente, entonces graba el nombre del archivo con su unidad
; y path delante y acabado en 0 en otro archivo llamado ENTRADA.ARC, que se
; coloca en el mismo path del programa, donde también está situado este módulo
; de compresión.
;
; - Este modulo de compresión tiene como misión abrir el archivo mencionado
; en ENTRADA.ARC y comprimirlo en el archivo de salida SALIDA.CMP, que queda
; listo para ser enviado por el puerto serie

                                        ;*******************************
                org     256             ;Programa

start           mov     ah,3ch          ;Función para crear archivo
                xor     cx,cx           ;Atributos a 0
                mov     dx,nameout      ;Nombre del archivo: SALIDA.CMP
                int     33              ;Crear archivo de salida

                jnc     noerror01       ;Si no hay error, continuar
                mov     ax,4c01h        ;Código de error 1
                int     33              ;Salir a Windows
noerror01
```

```
                mov     [handleout],ax  ;Salvar handle de SALIDA.CMP

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namepath     ;Nombre del archivo a abrir
                int     33              ;Abrir ENTRADA.ARC

                jnc     noerror02       ;Si no hay error, continuar
                mov     ax,4c02h        ;Código de error 2
                int     33              ;Salir a Windows
noerror02
                xchg    ax,bx           ;Handle de ENTRADA.ARC en bx
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,400          ;Número máximo de bytes a leer
                mov     dx,pathbuffer   ;Posición donde se escriben los bytes
                int     33              ;Leer bytes

                jnc     noerror03       ;Si no hay error, continuar
                mov     ax,4c03h        ;Código de error 3
                int     33              ;Salir a Windows
noerror03
                mov     si,dx           ;Apuntar al nombre de archivo
                mov     dx,1            ;Abrir archivo existente, no crear
                xor     bx,bx           ;Modo de acceso solo lectura
                xor     cx,cx           ;Sin atributos
                mov     ax,716ch        ;Función para crear o abrir archivo
                                        ; usando nombres largos
                int     33              ;Abrir archivo

                jnc     noerror05       ;Si no hay error, continuar
                mov     ax,4c05h        ;Código de error 5
                int     33              ;Salir a Windows

noerror05       mov     [handlein],ax   ;Salvar handle del archivo de entrada

                xchg    ax,bx           ;Handle en bx
                mov     ah,3fh          ;Función para leer
                mov     cx,3072         ;3K van a ser leídos
                mov     dx,buffer       ;En el buffer
                int     33              ;Leer
                call    copytoend       ;Copiar los primeros 128bytes al final

                add     dx,ax           ;Final de bytes leídos
                cmp     ax,cx           ;Si se leen menos de 6K
                jnc     nosmallfile
                mov     [EOF],dx        ;Marcar End of File
                jmp     short bucle     ;Y no poner readedge
nosmallfile     add     ax,buffer-128   ;Readedge > una vez procesado hasta
                mov     [readedge],ax   ; aquí, tenemos que leer 128 nuevos
                                        ; bytes
                ;La compresión comienza aquí

bucle           inc     word [maxdistance] ;Aumentar la distancia máxima de
                                        ;  búsqueda de coincidencia en 1 byte
                mov     si,[current]    ;Posición del byte actual
                cmp     si,[EOF]        ;Verificar si hemos llegado al final
                jnz     noEOFfound      ; del archivo, si no es así saltar
                mov     ax,[untagged_literals]
                and     ax,ax           ;Comprobar si quedan literales
                jz      nomoreflush     ; pendientes de marcar
                call    tagliterals     ;Marcarlos
nomoreflush     call    puttag1         ;Poner
                call    puttag0         ; código
                call    puttag0         ; de
                call    puttag0         ; fin
                call    puttag0         ; de
                call    puttag1         ; archivo
                call    puttag1         ; correspondiente
                call    puttag1         ; a 10000111
                call    puttag0         ;Rellenar con ceros para que se
                call    puttag0         ; escriba el último tag-byte y se
```

```
                call    puttag0         ; vacíen los literales que haya
                call    puttag0         ; pendientes
                call    puttag0         ;con 7 nos aseguramos de ello
                call    puttag0
                call    puttag0

                mov     ah,40h          ;Función de escritura
                mov     bx,[handleout]  ;Handle del archivo de salida
                mov     cx,[currentoutput] ;Puntero de escritura
                mov     dx,outbuffer    ;Dirección del buffer de escritura
                sub     cx,dx           ;Número de bytes a escribir
                int     33              ;Vaciar el buffer de escritura

end             mov     ax,4c00h        ;Sin error
                int     33              ;Salir a Windows


noEOFfound      push    si              ;Almacenar posición actual en la pila
                lodsb                   ;Leer byte
                cmp     si,endbuffer    ;Si la siguiente posición llega al
                jnz     nowrap00        ; final del buffer
                mov     si,buffer       ;colocarla al principio
nowrap00        mov     [current],si    ;actualizar posición de byte procesado
                cmp     si,[readedge]   ;si hemos llegado al límite de lectura
                jnz     noedgecrossed   ;hay que leer un nuevo bloque, si no
                                        ;se continúa

                pusha                   ;Guardar registros
                mov     cx,128          ;Bytes a leer / avanzar
                mov     dx,si           ;Se va a leer en
                add     dx,cx           ; la posición actual + 128
                cmp     dx,endbuffer    ;¿esto está al final del buffer?
                jnz     nowrap02        ;NO: saltar
                mov     dx,buffer       ;SI: colocar al comienzo
nowrap02        mov     ah,3fh          ;Función de lectura
                mov     bx,[handlein]   ;Handle del archivo de entrada
                int     33              ;Leer 128 bytes
                jnc     noerror04       ;Si no hay error, continuar
                mov     ax,4c04h        ;Código de error 4
                int     33              ;Salir a Windows
noerror04
                cmp     dx,buffer       ;Si hemos leido en el principio
                jnz     nocopytoend
                call    copytoend       ;Pasarlo al final también
nocopytoend
                cmp     ax,cx           ;Verificar si se han leido 128 bytes
                jz      EOFnotFOUND     ;Si es así saltar
EOFFOUND        add     dx,ax           ;Si se han leido menos, colocar
                mov     [EOF],dx        ; la marca de fin de archivo
                jmp     short NoReadEdge; y no fijar un nuevo readedge
EOFnotFOUND     mov     [readedge],dx   ;Actualizar readedge
NoReadEdge      sub     word [maxdistance],128 ;Actualizar maxdistance
                popa                    ;Restaurar registros
noedgecrossed   pop     si              ;Byte en al, si=current

                mov     bl,al
                mov     bh,0
                add     bx,bx           ;BX=AL*2

                mov     bp,[bx+prev]    ;Buscar última posición del byte en
                                        ; el buffer
                mov     [bx+prev],si    ;Actualizar última posición

                mov     cx,si           ;CX = distancia al previo, para
                sub     cx,bp           ; poner en prevbuff

                ja      nowrap03        ;Si la distancia es 0 o negativa
                add     cx,3072         ;Ajustarla
nowrap03                                ;CX=distancia entre si y bp
                mov     bx,si
```

```
                mov     [bx+si],cx      ;Actualizar prevbuff

                dec     byte[codepaired] ;Si el byte ya está comprimido en un
                jns     near bucle       ; codepair, no hace falta buscar
                inc     byte[codepaired] ; coincidencias

                and     bp,bp           ;Si bp es 0, el byte es un literal
                jz      near SETLITERAL_clearprevbuff ; colocarlo

                cmp     al,[bp]         ;Si el previo no coincide, era falso
                jnz     near SETLITERAL_clearprevbuff ;Por lo tanto > literal

                  mov     byte [max_match_lenght],0 ;Iniciar distancia máxima
hallada
                                        ; hasta el momento

        ;Entrada al bucle de búsqueda > Distancia acumulada en CX
        ;                             > Posición del siguiente a compara en BP
loop00          cmp     [maxdistance],cx ; si la distancia sobrepasa el límite
                jc      exitsearchloop  ;dejar de buscar

                push    si              ;Guardar posición actual

                mov     di,bp   ;SI=position_temp_curr
                                ;DI=position_temp_comp
                mov     dl,0    ;DL=Máxima longitud de coincidencia actual
keepmatching    cmpsb           ;Comparar bytes
                jnz     nomore  ;Si no coinciden, salir de esta comparación
                inc     dl      ;Si coinciden, incrementar número de bytes
                js      nomore  ;Si es 128, no seguir buscando
                cmp     si,[EOF];Si hemos llegado al final del archivo,
                jnz     keepmatching ; tampoco, en caso contrario seguir
nomore          cmp     [max_match_lenght],dl ;Comparar coincidencia con la
                                        ; máxima hasta ahora
                jnc     trynext         ;Si no la supera, pasar a la
                                        ; siguiente
                mov     [max_match_lenght],dl ;Si la supera, guardar la
                                        ; longitud
                mov     [max_match_distance],cx ; y la distancia del match
                cmp     dl,128          ;si es de longitud máxima
                jz      exitsearchmax   ; dejar de buscar

trynext         mov     di,bp           ;Para acceder a prevbuff[bp]
                add     cx,[bp+di]      ;Acumular distancia

                sub     bp,[bp+di]      ;Hallar nueva posición
                cmp     bp,buffer       ;Ver se pasa del buffer
                jnc     nowrap07        ;Si no, saltar
                add     bp,3072         ; Si se pasa, ajustar
nowrap07        pop     si              ;Recuperar posición actual
                jmp     short loop00    ;Repetir el bucle de búsqueda

exitsearchmax   pop     si              ;Limpiar pila
exitsearchloop  cmp     [max_match_lenght],byte 2  ;Si la distancia es menor
                jc      near SETLITERAL ; a 2, poner un literal
                jnz     setcodepair     ; si es mayor, poner un codepair
                cmp     [max_match_distance],word 255 ;Si es = 2, depende de
                jnc     SETLITERAL      ; la distancia. Mayor a 255 > literal

setcodepair       mov       cx,[untagged_literals] ;Ver si hay literales sin
marcar
                jcxz    notagliterals   ;Si no, saltar
                call    tagliterals     ;Marcar los literales pendientes
notagliterals   mov     al,[max_match_lenght] ;Fijar el número de
                dec     ax              ; bytes que no deben
                mov     [codepaired],al ; realizar la búsqueda

                call    puttag1         ;Codificar un 1
                mov     bx,[max_match_distance] ; BX=distancia
                bsr     cx,bx           ;Número de bits que van a ser
```

```
                                        ; codificados
                btr     bx,cx           ;Eliminar el bit mayor
                bt      cx,3            ;Colocar el bit 3 del número de bits
                call    puttagbit       ; de la distancia
                bt      cx,2            ;Excitar flag C según el bit 2
                call    puttagbit       ;Codificar bit
                bt      cx,1            ;Ahora el 1
                call    puttagbit       ;Codificar bit
                bt      cx,0            ;Y por último el 0
                call    puttagbit       ;Codificar bit
                jcxz    x_xdone         ;Si no hay que poner mas bits, la
                                        ; distancia ya está hecha
                shl     bx,1            ;Adaptar bx
putx_x          bt      bx,cx           ;Averiguar bit de la distancia
                call    puttagbit       ;Codificarlo
                loop    putx_x          ;Colocarlos todos
x_xdone         mov     bl,[max_match_lenght] ;
                xor     bh,bh           ;BX = longitud de la coincidencia
                dec     bx              ;Codificar uno menos, ya que el minimo
                                        ; es 2
                bsr     cx,bx           ;Número de bits a codificar
                btr     bx,cx           ;Eliminar el bit mayor
                bt      cx,2            ;Colocar los 3 bits del número de bits
                call    puttagbit       ; de la longitud
                bt      cx,1            ;Al igual que se hizo antes con los
                call    puttagbit       ; de la distancia
                bt      cx,0            ;Último bit...
                call    puttagbit       ;Codificarlo
                jcxz    y_ydone         ;Saltar si no hay bits que colocar
                shl     bx,1            ;Adaptar bx
puty_y          bt      bx,cx           ;Excitar flag C
                call    puttagbit       ;Codificar bit
                loop    puty_y          ;Hacerlo con todos
y_ydone         jmp     bucle           ;Volver al bucle principal

SETLITERAL_clearprevbuff
                mov     [bx+si],word 3071 ;Colocar en prevbuff una distancia
                                        ; tal que se pasa y no sigue buscando
SETLITERAL      mov     bx,[untagged_literals] ; Número de literales sin
                                        ; marcar
                and     bx,bx           ;Comprobar si es el primero
                jnz     nofirstuntagged ;Si no lo es saltar
                mov     [untagged_literal_start],si ; Si es el primero,
                                        ; colocar dirección de comienzo de
                                        ; los literales
nofirstuntagged inc     bx              ;Incrementar número de literales
                mov     [untagged_literals],bx ;Guardarlo
                cmp     bx,261          ;Comprobar si se ha llegado al máximo
                jnz     y_ydone         ;Si no, volver al bucle principal
                push    word bucle      ;Si se llega al máximo, ejecutar
                                        ; tagliterals y volver al bucle

tagliterals     pusha                   ;Guardar registros
                 mov        bx,[untagged_literals]  ;  Número  de  literales  sin
marcar
                mov     bp,bx           ;También en BP
                cmp     bx,7            ;Si es menor que 7
                jc      plain_literals  ;Ir a plain_literals
                call    puttag1         ;Si no
                call    puttag1         ;Colocar los 3 bits que indican
                call    puttag1         ; cadena de literales
                sub     bx,6            ;Restar 6 al número de bytes
                bsr     cx,bx           ; que va a ser codificado
                btr     bx,cx           ;Eliminar el bit mayor
                bt      cx,2
                call    puttagbit       ; Colocar el número de bits
                bt      cx,1            ;del número de bits de la cadena
                call    puttagbit
                bt      cx,0
                jcxz    lastbit         ;El último bit debe dejarse para el
```

```
                                        ; final
                call    puttagbit
                dec     cx              ;Decrementar cx
                jz      lastbit         ;El último bit debe dejarse para el
                                        ; final
puty            bt      bx,cx           ;Coger bit
                call    puttagbit       ;Codificarlo
                loop    puty            ;Repetir con todos

lastbit         mov     ax,[tagged_literals]    ;Ver si hay literales ya
                and     ax,ax           ; marcados
                jnz     noanotherstart  ;Si los hay, no cambiar el comienzo
                mov     dx,[untagged_literal_start] ;Coger el comienzo de los
                                        ; no marcados
                mov     [tagged_literal_start],dx ; Y colocarlo en el comienzo
                                        ; de los marcados
noanotherstart  add      [tagged_literals],bp     ;Incrementar número de bytes
marcados
                xor     ax,ax                   ;AX=0
                bt      bx,ax                   ;Último bit por colocar
                call    puttagbit               ;Codificarlo
                mov     [untagged_literals],ax  ;Literales sin marcar=0
                popa                            ;Restaurar registros
                ret                             ;Volver

plain_literals  mov     ax,[tagged_literals]    ;Ver si ya hay literales
                and     ax,ax           ; marcados
                jnz     noanotherstart2 ;Si los hay, no cambiar el comienzo
                mov     dx,[untagged_literal_start] ;No hay: Poner el comienzo
                                        ; de los no marcados
                mov     [tagged_literal_start],dx ; en el de los marcados
noanotherstart2 inc     word [tagged_literals]  ;Incrementar número de
                                        ; literales marcados
                call    puttag0                 ;Codificar un 0
                inc     word [untagged_literal_start] ;Incrementar comienzo de
                                        ; literales sin marcar
                dec     word [untagged_literals];Decrementar número de
                                        ; literales sin marcar
                jnz     plain_literals  ;Repetir mientras queden
                popa                    ;Restaurar registros
                ret                     ;Volver

copytoend       pusha                   ;Guardar registros
                mov     si,buffer       ;Origen: comienzo del buffer
                mov     di,endbuffer    ;Destino: final del buffer
                mov     cx,64           ;64 words
                rep     movsw           ;Copiar
                popa                    ;Restaurar registros
                ret                     ;Volver

puttag0         clc                     ;Flag C = 0
                jmp     short puttagbit ;Ir a poner el bit
puttag1         stc                     ;Flag C = 1
puttagbit       pusha                   ;Guardar registros
                mov     dl,[tag_bits]   ;Coger tag_bits actuales
                adc     dl,dl           ;Colocar nuevo bit
                jnc     nonexttagbyte   ;Si no sale el marker bit, saltar
                mov     al,dl           ;Si sale, colocar el al
                call    writebyte       ;Y escribir el byte
                mov     cx,[tagged_literals] ;Comprobar si hay literales
                                        ; marcados
                jcxz    notaggedliterals ; Si no, saltar
                mov     si,[tagged_literal_start] ;Si los hay, poner el origen
                                        ; en si
putliterals     lodsb                   ;Cargar literal marcado
                call    writebyte       ;Escribirlo
                cmp     si,endbuffer    ;Si llegamos al final del buffer
                jnz     nowrap08
                mov     si,buffer       ;Colocar al principio
nowrap08        loop    putliterals     ;Hacer todos los literales marcados
```

```
                mov     [tagged_literals],cx ;Ahora literales marcados=0
notaggedliterals mov     dl,1           ;Nuevo marker-bit
nonexttagbyte   mov     [tag_bits],dl   ;Actualizar tag_bits
                popa                    ;Restaurar registros
                ret                     ;Volver

writebyte       pusha                   ;Guardar registros
                mov     di,[currentoutput] ;Posición actual del puntero de
                                        ; escritura
                stosb                   ;Escribir el byte en memoria

                cmp     di,endoutbuff   ;¿El puntero llega al final?
                jnz     nolastbyte      ;NO: Saltar
                mov     ah,40h          ;Función de escritura
                mov     bx,[handleout]  ;Handle del archivo de salida
                mov     cx,32768        ;Número de bytes
                mov     dx,outbuffer    ;Buffer de escritura
                int     33              ;Escribir bytes

                jnc     noerror07       ;Si no hay error, continuar
                mov     ax,4c07h        ;Código de error 7
                int     33              ;Salir a Windows
noerror07
                mov     di,dx           ;Colocar puntero al principio
nolastbyte      mov     [currentoutput],di  ; Actualizar puntero de escritura
                popa                    ;Restaurar registros
                ret                     ;Volver

;FIN DEL PROGRAMA

                                        ;*******************************
                                        ;Variables con valores iniciados

EOF             dw      65535           ;(Basta con que esté fuera del buffer)


current         dw      buffer  ;Posición de memoria siendo procesada

namepath        db      "ENTRADA.ARC",0
nameout         db      "SALIDA.CMP",0

tagged_literal_start    dw    0       ;Comienzo de los literales marcados
tagged_literals         dw    0       ;Número de literales marcados
untagged_literal_start  dw    0       ;Comienzo de los literales sin marcar
untagged_literals       dw    0       ;Número de literales sin marcar


tag_bits        db      1               ;Tag byte en construcción

codepaired      db      0               ;Número de bytes a saltar

currentoutput   dw      outbuffer
maxdistance     dw      -1      ;Hasta cuanto podemos retroceder al buscar
                                ; un match


prev            times   256 dw 0


                                        ;*******************************
                                        ;*******************************
section .bss                            ;Variables sin iniciar


handleout       resw    1       ;Handle del archivo de salida
handlein        resw    1       ;Handle del archivo de entrada

max_match_lenght  resb    1     ;Máxima lóngitud de match hallado
max_match_distance resw    1    ;Distancia a la que se encuentra dicho match
```

```
readedge        resw    1       ;Límite de proceso hasta volver a leer un
                                ; bloque de 128 bytes


pathbuffer      resb    400     ;Aquí se almacena la posición del último
                                ; byte de un determinado tipo

buffer          EQU     8192
endbuffer       EQU     buffer+3072
prevbuff        EQU     16384
outbuffer       EQU     28672
endoutbuff      EQU     outbuffer+32768

;errorlevels:
;
; 0 - OK, no error
; 1 - No se puede crear SALIDA.CMP
; 2 - No se puede abrir ENTRADA.ARC
; 3 - No se puede leer ENTRADA.ARC
; 4 - Error de lectura
; 5 - No se puede abrir el archivo de entrada
; 7 - Error de escritura
```

8.2. CODECS DE DESCOMPRESIÓN

8.2.1. COPYUN.NAS

```
;COPYUN: Unpacker para el proyecto de fin de carrera
; (Método 3 - COPY > sin compresión)
;
; Realizado en lenguaje ensamblador, se compila con NASM
;
; - El programa maestro de Windows, recibe por el puerto serie el nombre del
; archivo y permite elegir la localización del directorio de salida
; visualmente. Entonces graba el nombre del archivo con su path y acabado en
; 0 en otro archivo llamado SALIDA.ARC, que se coloca en el mismo path del
; programa, donde también está situado este módulo de descompresión.
;
; - Este modulo de compresión tiene como misión abrir el archivo comprimido
; ENTRADA.CMP y descomprimir el archivo comprimido en el path y con el nombre
; que han sido indicados en SALIDA.ARC

                org     256

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namein       ;Nombre del archivo a abrir
                int     33              ;Abrir ENTRADA.CMP

                jnc     noerror01       ;Si no hay error, continuar
                mov     ax,4c01h        ;Código de error 1
                int     33              ;Salir a Windows

noerror01       mov     [handlein],ax   ;Salvar handle de ENTRADA.CMP

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namepath     ;Nombre del archivo a abrir
                int     33              ;Abrir SALIDA.ARC

                jnc     noerror02       ;Si no hay error, continuar
                mov     ax,4c02h        ;Código de error 2
                int     33              ;Salir a Windows

noerror02       xchg    ax,bx           ;Handle de SALIDA.ARC en bx
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,400          ;Número máximo de bytes a leer
                mov     dx,pathbuffer   ;Posición donde se escriben los bytes
                int     33              ;Leer bytes
```

```
                jnc     noerror03       ;Si no hay error, continuar
                mov     ax,4c03h        ;Código de error 3
                int     33              ;Salir a Windows

noerror03
                mov     si,dx           ;Apuntar al nombre de archivo con path
                mov     dx,18           ;Crear o truncar archivo existente
                mov     bx,1            ;Modo de acceso solo escritura
                xor     cx,cx           ;Sin atributos
                mov     ax,716ch        ;Función para crear o abrir archivo
                                        ; usando nombres largos
                int     33              ;Crear archivo de salida

                jnc     noerror05       ;Si no hay error, continuar
                mov     ax,4c05h        ;Código de error 5
                int     33              ;Salir a Windows

noerror05       mov     [handleout],ax  ;Salvar handle del archivo de salida

buclecopy       mov     ah,3fh
                mov     bx,[handlein]
                mov     cx,6000
                mov     dx,buffer
                int     33              ;Leer archivo

                jnc     noerror06
                mov     ax,4c06h        ;Código de error 6
                int     33              ;Salir a Windows
noerror06

                xchg    ax,cx           ;Número de bytes leídos en cx
                jcxz    exit            ;Si es 0, salir

                mov     ah,40h
                mov     bx,[handleout]
                int     33              ;Escribir archivo

                jnc     noerror07
                mov     ax,4c07h        ;Código de error 7
                int     33              ;Salir a Windows
noerror07
                jmp     buclecopy       ;Seguir copiando

exit            mov     ax,4c00h        ;Código de error 0 (sin problemas)
                int     33              ;Salir a Windows

namein          db      "ENTRADA.CMP",0
namepath        db      "SALIDA.ARC",0

section .bss

handleout       resw    1
handlein        resw    1
pathbuffer      resb    400
buffer          resb    6000

;errorlevels:
;
; 0 - OK, no error
; 1 - No se puede abrir ENTRADA.CMP
; 2 - No se puede abrir SALIDA.ARC
; 3 - No se puede leer SALIDA.ARC
; 5 - No se puede crear el archivo de salida
; 6 - Error de lectura
```

8.2.2. LZ6KUN.NAS

```
;LZ6KUN: Unpacker para el proyecto de fin de carrera
; (Método 1 - LZ6K > LZ greedy con diccionario de 6K)
;
; Versión 1.0
;
; Realizado en lenguaje ensamblador, se compila con NASM
;
; - El programa maestro de Windows, recibe por el puerto serie el nombre del
; archivo y permite elegir la localización del directorio de salida
; visualmente. Entonces graba el nombre del archivo con su path y acabado en
; 0 en otro archivo llamado SALIDA.ARC, que se coloca en el mismo path del
; programa, donde también está situado este módulo de descompresión.
;
; - Este modulo de compresión tiene como misión abrir el archivo comprimido
; ENTRADA.CMP y descomprimir el archivo comprimido en el path y con el nombre
; que han sido indicados en SALIDA.ARC

                org     256

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namein       ;Nombre del archivo a abrir
                int     33              ;Abrir ENTRADA.CMP

                jnc     noerror01       ;Si no hay error, continuar
                mov     ax,4c01h        ;Código de error 1
                int     33              ;Salir a Windows
noerror01
                mov     [handlein],ax   ;Salvar handle de ENTRADA.CMP

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namepath     ;Nombre del archivo a abrir
                int     33              ;Abrir SALIDA.ARC

                jnc     noerror02       ;Si no hay error, continuar
                mov     ax,4c02h        ;Código de error 2
                int     33              ;Salir a Windows
noerror02
                xchg    ax,bx           ;Handle de SALIDA.ARC en bx
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,400          ;Número máximo de bytes a leer
                mov     dx,pathbuffer   ;Posición donde se escriben los bytes
                int     33              ;Leer bytes

                jnc     noerror03       ;Si no hay error, continuar
                mov     ax,4c03h        ;Código de error 3
                int     33              ;Salir a Windows
noerror03
                mov     si,dx           ;Apuntar al nombre de archivo con path
                mov     dx,18           ;Crear o truncar archivo existente
                mov     bx,1            ;Modo de acceso solo escritura
                xor     cx,cx           ;Sin atributos
                mov     ax,716ch        ;Función para crear o abrir archivo
                                        ; usando nombres largos
                int     33              ;Crear archivo de salida

                jnc     noerror05       ;Si no hay error, continuar
                mov     ax,4c05h        ;Código de error 5
                int     33              ;Salir a Windows
noerror05
                mov     [handleout],ax  ;Salvar handle del archivo de salida

                mov     dx,inbuff       ;Posición en la que se lee
                mov     cx,32768        ;Bytes a leer
                mov     bx,[handlein]   ;Handle del archivo de entrada
                mov     ah,3fh          ;Función de lectura
                int     33              ;Leer hasta 32K de datos comprimidos
```

```
                mov     si,dx           ;Puntero de lectura
                mov     di,outbuff      ;Puntero de escritura

                mov     dl,128          ;Marker bit
bucle           xor     bp,bp           ;Número de bytes que se retroceden
                                        ; en el puntero de lectura al
                                        ; escribir 128 bytes = 0
                call    getbit          ;Tomar un bit
                jnc     literalbyte     ;Si es 0, ir a literalbyte
                mov     cx,4            ;Si no, prepararse para leer 4 bits
                xor     bx,bx           ;Borrar registro de destino
                call    getbits         ;Leer 4 bits en bx
                cmp     bx,13           ;Si el resultado es < 13
                jc      codepair        ; es un codepair
                                        ;Si no, es una cadena de litarales
literalchain    and     bx,3            ;Guardar los dos últimos bits
                inc     cx              ;Completar YYY con otro bit
                call    getbits         ;Leerlo
                inc     cx              ;Primer bit del número de bytes - 4
                xchg    bx,cx           ;Se leen cx bits en bx
                call    getbits         ;bx=numero de bytes encadenados - 4
                add     bx,4            ;Completar número de bytes
                mov     cx,bx           ;Colocar en cx
chain           call    get_si          ;Tomar un literal byte de la cadena
                call    put_di          ;Colocarlo en el buffer de salida
                loop    chain           ;Repetir hasta que cx=0
                jmp     bucle           ;Seguir descomprimiendo

literalbyte     call    get_si          ;Tomar un literal
                call    put_di          ;Colocarlo en el buffer de salida
                jmp     bucle           ;Seguir descomprimiendo

codepair        inc     cx              ;Primer bit de la distancia
                xchg    bx,cx           ;Se leen cx bits en bx
                call    getbits         ;Leer distancia en bx
                push    bx              ;Guardar distancia en la pila
                mov     cl,3            ;Leer 3 bits
                xor     bx,bx           ;Borrar registro de destino
                call    getbits         ;Leer número de bits de la longitud
                inc     cx              ;Primer bit de la longitud
                cmp     bx,7            ;Si el número es 7, hemos alcanzado
                jz      EOF             ; el final del archivo > salir
                xchg    bx,cx           ;Se leen cx bits en bx
                call    getbits         ;bx=numero de bytes repetidos - 1
                inc     bx              ;Completar longitud de la coincidencia
                mov     cx,bx           ;Colocar en cx
                pop     bx              ;Distancia en bx

                push    si              ;Guardar si
                mov     si,di           ;Hacer que si apunte a la parte que
                sub     si,bx           ; va a repetirse, = puntero de
                                        ; escritura - distancia

                mov     bp,128          ;Número de bytes que se retroceden
                                        ; en el puntero de lectura al
                                        ; escribir 128 bytes

uncomp_match    lodsb                   ;Leer un byte
                call    put_di          ;Escribirlo
                loop    uncomp_match    ;Repetir cx veces
                pop     si              ;Restaurar si
                jmp     bucle           ;Seguir descomprimiendo

EOF             pop     bx              ;(sacar distancia de la pila)
                mov     ah,40h          ;Función para escribir en archivo
                mov     dx,outbuff      ;Escribir desde el comienzo del buffer
                mov     cx,di           ;Todos los bytes que haya
                sub     cx,dx           ;CX = DI-DX
                mov     bx,[handleout]  ;Handle del archivo de salida
                int     33              ;Vaciar buffer de escritura
```

```
exit            mov     ax,4c00h        ;Código de error 0 (sin problemas)
                int     33              ;Salir a Windows

getbits         jcxz    nobitstotake    ;Si no hay bits que tomar, saltar
getbitsl        call    getbit          ;Tomar un bit
                adc     bx,bx           ;Colocarlo en bx
                loop    getbitsl        ;Repetir
nobitstotake    ret

getbit          add     dl,dl           ;Sacar bit por el flag C
                jnz     gotten          ;Si el tag-byte es 0, hay que
                xchg    ax,dx           ;  tomar otro
                call    get_si          ;Leer byte en AL
                xchg    ax,dx           ;  pasar byte a DL
                stc                     ;Flag C para poner el bit de marca
                adc     dl,dl           ;Sacar bit y poner la marca
gotten          ret

get_si          cmp     si,inbuffend    ;¿Estamos en el borde?
                jnz     dontread        ;  si no: no leer
                pusha                   ;Guardar registros
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,32768        ;Número máximo de bytes a leer
                mov     bx,[handlein]   ;Handle del archivo comprimido
                mov     dx,inbuff       ;Leer en el buffer de entrada
                int     33              ;Leer hasta otros 32K de datos

                jnc     noerror04       ;Si no hay error, continuar
                mov     ax,4c04h        ;Código de error 4
                int     33              ;Salir a Windows
noerror04
                popa
                mov     si,inbuff       ;Colocar puntero al comienzo de los
                                        ; nuevos datos
dontread        lodsb                   ;Leer un byte
                ret

put_di          stosb                   ;Guardar byte en el diccionario
                cmp     di,outbuffend   ;Si no llegamos al tope
                jnz     dontwrite       ;No escribir nada

                pusha                   ;Hemos llegado al final del buffer
                mov     dx,outbuff      ;Escribir la primera parte de los
                mov     cx,128          ; datos descomprimidos, 128 bytes
                mov     ah,40h          ;Función para escribir archivo
                mov     bx,[handleout]  ;Handle del archivo de salida
                int     33              ;Escribir

                jnc     noerror06       ;Si no hay error, continuar
                mov     ax,4c06h        ;Código de error 6
                int     33              ;Salir a Windows
noerror06
                mov     di,dx           ;Destino
                mov     si,outbuff+128  ;Origen
                mov     cx,3008         ;Número de bytes a mover / 2
                rep     movsw           ;Mover al comienzo del buffer
                popa                    ;Restaurar registros
                sub     di,128          ;Retroceder puntero de escritura
                sub     si,bp           ;Y el de lectura solo si es necesario
dontwrite       ret

namein          db      "ENTRADA.CMP",0
namepath        db      "SALIDA.ARC",0

section .bss

handleout       resw    1
handlein        resw    1
pathbuffer      resb    400
```

```
outbuff         EQU     8192
outbuffend      EQU     outbuff+6144
inbuff          EQU     16384
inbuffend       EQU     inbuff+32768

;errorlevels:
;
; 0 - OK, no error
; 1 - No se puede abrir ENTRADA.CMP
; 2 - No se puede abrir SALIDA.ARC
; 3 - No se puede leer SALIDA.ARC
; 4 - Error de lectura
; 5 - No se puede crear el archivo de salida
; 6 - Error de escritura

;CODIFICACIÓN:
;0 - Byte literal
;       Se copia el siguiente byte.
;
;1XXXX - Repetición de bloque previo.
;       (XXXX de 0000 hasta 1100) - se toman XXXX bits para formar la
;       distancia (poniendole un uno delante), los siguientes 3 bits forman
;       YYY, y se toman YYY bits para formar la longitud (se pone un uno
;       delante y se le suma 1). Si YYY=111, fin de archivo
;Rango: distancia > 1 a 8191 (se usa hasta 6016).
;       longitud  > 2 a 128
;
;111YYY - Cadena de bytes literales.
;       (YYY de 010 a 111) - se toman YYY bits para formar el número de bytes
;       que se copian directamente, poniendo uno delante y sumando 4.
```

8.2.3. LZ3KUN.NAS

```
;LZ3KUN: Unpacker para el proyecto de fin de carrera
; (Método 2 - LZ3K > LZ greedy con diccionario de 3K)
;
; Versión 1.0
;
; Realizado en lenguaje ensamblador, se compila con NASM
;
; - El programa maestro de Windows, recibe por el puerto serie el nombre del
; archivo y permite elegir la localización del directorio de salida
; visualmente. Entonces graba el nombre del archivo con su path y acabado en
; 0 en otro archivo llamado SALIDA.ARC, que se coloca en el mismo path del
; programa, donde también está situado este módulo de descompresión.
;
; - Este modulo de compresión tiene como misión abrir el archivo comprimido
; ENTRADA.CMP y descomprimir el archivo comprimido en el path y con el nombre
; que han sido indicados en SALIDA.ARC

                org     256

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namein       ;Nombre del archivo a abrir
                int     33              ;Abrir ENTRADA.CMP

                jnc     noerror01       ;Si no hay error, continuar
                mov     ax,4c01h        ;Código de error 1
                int     33              ;Salir a Windows
noerror01
                mov     [handlein],ax   ;Salvar handle de ENTRADA.CMP

                mov     ax,3d00h        ;Función para abrir archivo
                mov     dx,namepath     ;Nombre del archivo a abrir
                int     33              ;Abrir SALIDA.ARC
```

```
                jnc     noerror02       ;Si no hay error, continuar
                mov     ax,4c02h        ;Código de error 2
                int     33              ;Salir a Windows
noerror02
                xchg    ax,bx           ;Handle de SALIDA.ARC en bx
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,400          ;Número máximo de bytes a leer
                mov     dx,pathbuffer   ;Posición donde se escriben los bytes
                int     33              ;Leer bytes

                jnc     noerror03       ;Si no hay error, continuar
                mov     ax,4c03h        ;Código de error 3
                int     33              ;Salir a Windows
noerror03
                mov     si,dx           ;Apuntar al nombre de archivo con path
                mov     dx,18           ;Crear o truncar archivo existente
                mov     bx,1            ;Modo de acceso solo escritura
                xor     cx,cx           ;Sin atributos
                mov     ax,716ch        ;Función para crear o abrir archivo
                                        ; usando nombres largos
                int     33              ;Crear archivo de salida

                jnc     noerror05       ;Si no hay error, continuar
                mov     ax,4c05h        ;Código de error 5
                int     33              ;Salir a Windows
noerror05
                mov     [handleout],ax  ;Salvar handle del archivo de salida

                mov     dx,inbuff       ;Posición en la que se lee
                mov     cx,32768        ;Bytes a leer
                mov     bx,[handlein]   ;Handle del archivo de entrada
                mov     ah,3fh          ;Función de lectura
                int     33              ;Leer hasta 32K de datos comprimidos

                mov     si,dx           ;Puntero de lectura
                mov     di,outbuff      ;Puntero de escritura

                mov     dl,128          ;Marker bit
bucle           xor     bp,bp           ;Número de bytes que se retroceden
                                        ; en el puntero de lectura al
                                        ; escribir 128 bytes = 0
                call    getbit          ;Tomar un bit
                jnc     literalbyte     ;Si es 0, ir a literalbyte
                mov     cx,4            ;Si no, prepararse para leer 4 bits
                xor     bx,bx           ;Borrar registro de destino
                call    getbits         ;Leer 4 bits en bx
                cmp     bx,12           ;Si el resultado es < 13
                jc      codepair        ; es un codepair
                                        ;Si no, es una cadena de litarales
literalchain    and     bx,3            ;Guardar los dos últimos bits
                inc     cx              ;Completar YYY con otro bit
                call    getbits         ;Leerlo
                inc     cx              ;Primer bit del número de bytes - 6
                xchg    bx,cx           ;Se leen cx bits en bx
                call    getbits         ;bx=numero de bytes encadenados - 6
                add     bx,6            ;Completar número de bytes
                mov     cx,bx           ;Colocar en cx

chain           call    get_si          ;Tomar un literal byte de la cadena
                call    put_di          ;Colocarlo en el buffer de salida
                loop    chain           ;Repetir hasta que cx=0
                jmp     bucle           ;Seguir descomprimiendo

literalbyte     call    get_si          ;Tomar un literal
                call    put_di          ;Colocarlo en el buffer de salida
                jmp     bucle           ;Seguir descomprimiendo

codepair        inc     cx              ;Primer bit de la distancia
                xchg    bx,cx           ;Se leen cx bits en bx
                call    getbits         ;Leer distancia en bx
```

```
                push    bx              ;Guardar distancia en la pila
                mov     cl,3            ;Leer 3 bits
                xor     bx,bx           ;Borrar registro de destino
                call    getbits         ;Leer número de bits de la longitud
                inc     cx              ;Primer bit de la longitud
                cmp     bx,7            ;Si el número es 7, hemos alcanzado
                jz      EOF             ; el final del archivo > salir
                xchg    bx,cx           ;Se leen cx bits en bx
                call    getbits         ;bx=numero de bytes repetidos - 1
                inc     bx              ;Completar longitud de la coincidencia
                mov     cx,bx           ;Colocar en cx
                pop     bx              ;Distancia en bx

                push    si              ;Guardar si
                mov     si,di           ;Hacer que si apunte a la parte que
                sub     si,bx           ; va a repetirse, = puntero de
                                        ; escritura - distancia
                mov     bp,128          ;Número de bytes que se retroceden
                                        ; en el puntero de lectura al
                                        ; escribir 128 bytes

uncomp_match    lodsb                   ;Leer un byte
                call    put_di          ;Escribirlo
                loop    uncomp_match    ;Repetir cx veces
                pop     si              ;Restaurar si
                jmp     bucle           ;Seguir descomprimiendo

EOF             pop     bx              ;(restaurar pila)
                mov     ah,40h          ;Función para escribir en archivo
                mov     dx,outbuff      ;Escribir desde el comienzo del buffer
                mov     cx,di           ;Todos los bytes que haya
                sub     cx,dx           ;CX = DI-DX
                mov     bx,[handleout]  ;Handle del archivo de salida
                int     33              ;Vaciar buffer de escritura

exit            mov     ax,4c00h        ;Código de error 0 (sin problemas)
                int     33              ;Salir a Windows


getbits         jcxz    nobitstotake    ;Si no hay bits que tomar, saltar
getbitsl        call    getbit          ;Tomar un bit
                adc     bx,bx           ;Colocarlo en bx
                loop    getbitsl        ;Repetir
nobitstotake    ret

getbit          add     dl,dl           ;Sacar bit por el flag C
                jnz     gotten          ;Si el tag-byte es 0, hay que
                xchg    ax,dx           ;  tomar otro
                call    get_si          ;Leer byte en AL
                xchg    ax,dx           ;  pasar byte a DL
                stc                     ;Flag C para poner el bit de marca
                adc     dl,dl           ;Sacar bit y poner la marca
gotten          ret

get_si          cmp     si,inbuffend    ;¿Estamos en el borde?
                jnz     dontread        ;  si no: no leer
                pusha                   ;Guardar registros
                mov     ah,3fh          ;Función para leer de archivo
                mov     cx,32768        ;Número máximo de bytes a leer
                mov     bx,[handlein]   ;Handle del archivo comprimido
                mov     dx,inbuff       ;Leer en el buffer de entrada
                int     33              ;Leer hasta otros 32K de datos

                jnc     noerror04       ;Si no hay error, continuar
                mov     ax,4c04h        ;Código de error 4
                int     33              ;Salir a Windows
noerror04
                popa
                mov     si,inbuff       ;Colocar puntero al comienzo de los
                                        ; nuevos datos
```

```
dontread        lodsb                   ;Leer un byte
                ret

put_di          stosb                   ;Guardar byte en el diccionario
                cmp     di,outbuffend   ;Si no llegamos al tope
                jnz     dontwrite       ;No escribir nada

                pusha                   ;Hemos llegado al final del buffer
                mov     dx,outbuff      ;Escribir la primera parte de los
                mov     cx,128          ; datos descomprimidos, 128 bytes
                mov     ah,40h          ;Función para escribir archivo
                mov     bx,[handleout]  ;Handle del archivo de salida
                int     33              ;Escribir

                jnc     noerror06       ;Si no hay error, continuar
                mov     ax,4c06h        ;Código de error 6
                int     33              ;Salir a Windows
noerror06
                mov     di,dx           ;Destino
                mov     si,outbuff+128  ;Origen
                mov     cx,1504         ;Número de bytes a mover / 2
                rep     movsw           ;Mover al comienzo del buffer
                popa                    ;Restaurar registros
                sub     di,128          ;Retroceder puntero de escritura
                sub     si,bp           ;Y el de lectura solo si es necesario
dontwrite       ret

namein          db      "ENTRADA.CMP",0
namepath        db      "SALIDA.ARC",0

section .bss

handleout       resw    1               ;Handle del archivo de salida
handlein        resw    1               ;Handle del archivo de entrada
pathbuffer      resb    400             ;Buffer para almacenar el path y
                                        ; nombre del archivo de salida


outbuff         EQU     8192
outbuffend      EQU     outbuff+3072
inbuff          EQU     16384
inbuffend       EQU     inbuff+32768

;errorlevels:
;
; 0 - OK, no error
; 1 - No se puede abrir ENTRADA.CMP
; 2 - No se puede abrir SALIDA.ARC
; 3 - No se puede leer SALIDA.ARC
; 4 - Error de lectura
; 5 - No se puede crear el archivo de salida
; 6 - Error de escritura

;CODIFICACIÓN:
;0 - Byte literal
;       Se copia el siguiente byte.
;
;1XXXX - Repetición de bloque previo.
;       (XXXX de 0000 hasta 1011) - se toman XXXX bits para formar la
;       distancia (poniendole un uno delante), los siguientes 3 bits forman
;       YYY, y se toman YYY bits para formar la longitud (se pone un uno
;       delante y se le suma 1). Si YYY=111, fin de archivo
;Rango: distancia > 1 a 4095 (se usa hasta 2944).
;       longitud  > 2 a 128
;
;111YYY - Cadena de bytes literales.
;       (YYY de 000 a 111) - se toman YYY bits para formar el número de bytes
;       que se copian directamente, poniendo uno delante y sumando 7.
```

8.3. PROGRAMA EN VISUAL BASIC

Proyecto.frm:

```
VERSION 5.00
Object = "{F9043C88-F6F2-101A-A3C9-08002B2F49FB}#1.2#0"; "COMDLG32.OCX"
Object = "{648A5603-2C6E-101B-82B6-000000000014}#1.1#0"; "MSCOMM32.OCX"
Begin VB.Form Form1
   BorderStyle     =   3  'Fixed Dialog
   Caption         =   "Jaime Tejedor Gómez --- Proyecto de Fin de Carrera ---
2001"
   ClientHeight    =   5280
   ClientLeft      =   45
   ClientTop       =   330
   ClientWidth     =   7440
   LinkTopic       =   "Form1"
   MaxButton       =   0   'False
   ScaleHeight     =   5280
   ScaleWidth      =   7440
   StartUpPosition =   3  'Windows Default
   Begin MSComDlg.CommonDialog CommonDialog1
      Left            =   1200
      Top             =   4080
      _ExtentX        =   847
      _ExtentY        =   847
      _Version        =   393216
      CancelError     =   -1  'True
      DialogTitle     =   "Guardar Como"
   End
   Begin MSCommLib.MSComm MSComm1
      Left            =   5280
      Top             =   3840
      _ExtentX        =   1005
      _ExtentY        =   1005
      _Version        =   393216
      DTREnable       =   -1  'True
      Handshaking     =   2
      InBufferSize    =   16384
      OutBufferSize   =   8192
      RThreshold      =   1
      RTSEnable       =   -1  'True
      BaudRate        =   115200
      SThreshold      =   1
   End
   Begin VB.DriveListBox Drive1
      Height          =   315
      Left            =   120
      TabIndex        =   13
      Top             =   1920
      Width           =   1815
   End
   Begin VB.FileListBox File1
      Height          =   2430
      Hidden          =   -1  'True
      Left            =   3480
      System          =   -1  'True
      TabIndex        =   12
      Top             =   2400
      Width           =   3855
   End
   Begin VB.DirListBox Dir1
      Height          =   2340
      Left            =   120
      OLEDropMode     =   1  'Manual
      TabIndex        =   11
      Top             =   2400
      Width           =   3255
   End
```

```
Begin VB.CommandButton Command3
   Caption         =   "Conectar"
   Height          =   255
   Left            =   2280
   TabIndex        =   10
   Top             =   840
   Width           =   1095
End
Begin VB.CommandButton Command2
   Caption         =   "Transmitir"
   Height          =   375
   Left            =   2160
   TabIndex        =   9
   Top             =   1920
   Width           =   1215
End
Begin VB.Frame Frame2
   Caption         =   "Puerto:"
   Height          =   735
   Left            =   2280
   TabIndex        =   3
   Top             =   1080
   Width           =   1095
   Begin VB.OptionButton COM2
      Caption         =   "COM2"
      Height          =   195
      Left            =   120
      TabIndex        =   8
      Top             =   480
      Width           =   855
   End
   Begin VB.OptionButton COM1
      Caption         =   "COM1"
      Height          =   195
      Left            =   120
      TabIndex        =   7
      Top             =   240
      Value           =   -1  'True
      Width           =   855
   End
End
Begin VB.TextBox Text1
   Height          =   285
   Left            =   0
   TabIndex        =   1
   Top             =   360
   Width           =   7335
End
Begin VB.Frame Frame1
   Caption         =   "Método:"
   Height          =   1095
   Left            =   0
   TabIndex        =   0
   Top             =   720
   Width           =   2055
   Begin VB.OptionButton METODO3
      Caption         =   "Sin Compresión"
      Height          =   255
      Left            =   120
      TabIndex        =   6
      Top             =   720
      Width           =   1695
   End
   Begin VB.OptionButton METODO2
      Caption         =   "LZ, diccionario 3K"
      Height          =   255
      Left            =   120
      TabIndex        =   5
      Top             =   480
      Width           =   1815
```

```
      End
      Begin VB.OptionButton METODO1
         Caption         =   "LZ, diccionario 6K"
         Height          =   195
         Left            =   120
         TabIndex        =   4
         Top             =   240
         Value           =   -1  'True
         Width           =   1815
      End
   End
   Begin VB.Label Label8
      Height          =   255
      Left            =   3480
      TabIndex        =   20
      Top             =   2040
      Width           =   3735
   End
   Begin VB.Label Label7
      Height          =   255
      Left            =   3480
      TabIndex        =   19
      Top             =   1800
      Width           =   3855
   End
   Begin VB.Label Label6
      Height          =   255
      Left            =   3480
      TabIndex        =   18
      Top             =   1560
      Width           =   3735
   End
   Begin VB.Label Label5
      Height          =   255
      Left            =   3480
      TabIndex        =   17
      Top             =   1320
      Width           =   3735
   End
   Begin VB.Label Label4
      Height          =   255
      Left            =   3480
      TabIndex        =   16
      Top             =   1080
      Width           =   3735
   End
   Begin VB.Label Label3
      Height          =   255
      Left            =   3480
      TabIndex        =   15
      Top             =   840
      Width           =   3735
   End
   Begin VB.Label Label1
      Caption         =   "ESTADO: Sin conexión"
      Height          =   255
      Left            =   120
      TabIndex        =   14
      Top             =   4920
      Width           =   2415
   End
   Begin VB.Label Label2
      Caption         =   "Archivo a transmitir:"
      Height          =   255
      Left            =   0
      TabIndex        =   2
      Top             =   120
      Width           =   1695
   End
End
```

```
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
'Funciones para hacer la llamada a los programas MSDOS
' que se usan para la compresión y decompresión
Private Declare Function GetActiveWindow Lib "User32" () As Long
Private Declare Function IsWindow Lib "User32" (ByVal hwnd As Long) As Long
Private Declare Function GetForegroundWindow Lib "User32" () As Long

'Variables globales

Dim PackMethod As Integer       'Método de compresión seleccionado
Dim UnpackMethod As Integer     'Método de descompresión del archivo recibido
Dim FileNameSend As String      'Nombre del archivo que se envia
Dim State As Integer            'Estado de recepción de datos
Dim Connected As Boolean        'Indicación de conexión
Dim WorkDrive As Variant        'Unidad de trabajo
Dim WorkDir As Variant          'Directorio de trabajo

Private Sub COM1_Click()
If Connected = False Then
MSComm1.CommPort = 1
End If
End Sub
Private Sub COM2_Click()
If Connected = False Then
MSComm1.CommPort = 2
End If
End Sub

Private Sub Command2_Click()
Dim vr As Long
Dim hWndAct As Long
Dim PackName As String
Dim Temp As String
Dim Zero As Byte

If (Text1.Text = "" Or (Dir(Text1.Text) = "")) Then
  MsgBox "Error: No hay archivo seleccionado", vbExclamation, "Error"
  Exit Sub
End If

If Connected = False Then
     MsgBox "Error: No puedes transmitir sin estar conectado", vbExclamation,
"Error"
    Exit Sub
End If

Zero = 0
BackupDrive = Drive1.Drive  'Salvar unidad
BackupPath = Dir1.Path      ' y directorio
Drive1.Drive = WorkDrive    'Colocarse en la unidad del programa
Dir1.Path = WorkPath        ' y en el directorio
Command2.Enabled = False    'Desactivar botón

Open "ENTRADA.ARC" For Binary As #2 'Crear el archivo de entrada
Put #2, , Text1.Text        'Colocando el nombre del archivo
Put #2, , Zero              'Y terminando con un cero
Close #2                    'Cerrar archivo ENTRADA.ARC

Drive1.Drive = BackupDrive  'Volver a colocarse la unidad
Dir1.Path = BackupPath      ' y directorio anteriores

Label3.Caption = "Comprimiendo"
Label4.Caption = "Tamaño: " + Str(FileLen(Text1.Text))
Label5.Caption = ""

Select Case PackMethod
```

```
    Case 1
    PackName = "LZ6KPA.COM"
    Case 2
    PackName = "LZ3KPA.COM"
    Case 3
    PackName = "COPYPA.COM"
End Select

vr = Shell(PackName, vbMinimizedFocus) 'Ejecutar programa en ventana MSDOS
minimizada
Do While GetActiveWindow() = hWndAct 'Esperar a que se abra la ventana
  vr = DoEvents()
Loop
hWndAct = GetForegroundWindow() 'Tomar ventana
Do While IsWindow(hWndAct)      'Esperar a que se cierre la ventana
  vr = DoEvents()               ' lo cual indicará que el programa
Loop                            ' compresor ha terminado

If (Dir("SALIDA.CMP") = "") Then
    MsgBox "Error: No se ha creado el archivo de salida", vbExclamation,
"Error"
    Label3.Caption = ""
    Kill "ENTRADA.ARC"
    Exit Sub
End If

Kill "ENTRADA.ARC"
Label3.Caption = "Compresión realizada, enviando"
Label5.Caption = "Tamaño comprimido: " + Str(FileLen("SALIDA.CMP"))

Select Case PackMethod
    Case 1
    MSComm1.Output = Chr(&H20)
    Case 2
    MSComm1.Output = Chr(&H21)
    Case 3
    MSComm1.Output = Chr(&H22)
End Select

FileLenght = FileLen("SALIDA.CMP")

D = FileLenght Mod 256
MSComm1.Output = Chr(D)
FileLenght = Int(FileLenght / 256)
D = FileLenght Mod 256
MSComm1.Output = Chr(D)
FileLenght = Int(FileLenght / 256)
D = FileLenght Mod 256
MSComm1.Output = Chr(D)
FileLenght = Int(FileLenght / 256)
D = FileLenght Mod 256
MSComm1.Output = Chr(D)

MSComm1.Output = FileNameSend + Chr(0)

Open "SALIDA.CMP" For Binary Access Read As #4
    ' Lee el archivo en bloques del tamaño del búfer de transmisión.
Bsize = MSComm1.OutBufferSize
LF& = LOF(4)    'Hallar tamaño del archivo
Do Until EOF(4)
    ' No lee demasiado al final.
    If LF& - Loc(4) <= Bsize Then
        Bsize = LF& - Loc(4)
    End If
    ' Lee un bloque de datos.
    Temp = Space$(Bsize)
    Get #4, , Temp
    ' Transmite el bloque.
    If Temp <> "" Then MSComm1.Output = Temp
    Label3.Caption = "Enviando " + Str(Loc(4)) + " bytes"
```

```
    If Err Then
        MsgBox Error$, 48
        Exit Do
    End If
   ' Espera a que se envíen todos los datos.
    Do
        Ret = DoEvents()
    Loop Until MSComm1.OutBufferCount = 0

    If LF& = Loc(4) Then Get #4, , Temp ' Quita el EOF sin enviarlo
    Loop
Close #4
Label3.Caption = "Archivo Enviado"
Kill "SALIDA.CMP"
Command2.Enabled = True    'Desactivar botón
End Sub
Private Sub Command3_Click()
Dim Ping As Byte
On Error Resume Next
Ping = &H31
If Connected = False Then
    MSComm1.PortOpen = True
    If (MSComm1.PortOpen = False) Then
         MsgBox "Error: El Puerto de comunicación está en uso", vbExclamation,
"Error"
        Exit Sub
    End If
    Command3.Caption = "Desconectar"
    Connected = True
    COM1.Enabled = False
    COM2.Enabled = False
    RThreshold = 1
    Label1.Caption = "ESTADO: Conectado"
    State = 1
    MSComm1.InputLen = 1
    MSComm1.Output = Chr(Ping)
Else
    MSComm1.PortOpen = False
    Connected = False
    Command3.Caption = "Conectar"
    COM1.Enabled = True
    COM2.Enabled = True
    Label1.Caption = "ESTADO: Sin Conexión"
End If
End Sub

Private Sub Dir1_Change()
 File1.Path = Dir1.Path
End Sub
Private Sub Drive1_Change()
On Error GoTo DriverError
  Dir1.Path = Drive1.Drive
  GoTo NoDriverError
DriverError:
  MsgBox "Error: Unidad no preparada", vbExclamation, "Error"
  Exit Sub
NoDriverError:
End Sub
Private Sub File1_Click()
If Right(Dir1.Path, 1) = "\" Then
    Text1.Text = Dir1.Path & File1.FileName
Else
    Text1.Text = Dir1.Path & "\" & File1.FileName
End If
FileNameSend = File1.FileName
End Sub
Private Sub Form_Load()
WorkDrive = Drive1.Drive
WorkDir = CurDir
Connected = False
```

```
PackMethod = 1
State = 1
End Sub


Private Sub METODO1_Click()
    PackMethod = 1
End Sub

Private Sub METODO2_Click()
    PackMethod = 2
End Sub

Private Sub METODO3_Click()
    PackMethod = 3
End Sub

Public Sub MSComm1_OnComm()
Static Longitud As Long
Static Longitud2 As Long
Static RecFileName As String
Dim RecFilePath As String
Dim RecFileAll As String
Dim Rec As String
Dim D As Long
Dim vr2 As Long
Dim hWndAct2 As Long
Dim UnpackName As String

Select Case MSComm1.CommEvent
    Case comEvReceive
        Select Case State
            Case 0  'Receiving a file of Longitud bytes
                Rec = MSComm1.Input
                Longitud = Longitud - Len(Rec)
                 Label6.Caption="Recibiendo archivo. Recibidos  "+Str(Longitud2
- Longitud) + " bytes"
                Print #1, Rec;
                If Longitud = 0 Then
                    Close #1
                    On Error GoTo NoGrabar
                    CommonDialog1.FileName = RecFileName
                    CommonDialog1.ShowSave

                    RecFileName = CommonDialog1.FileName

                    RecFileAll = RecFileName + Chr(0)
                    ChDrive (WorkDrive)
                    ChDir (WorkDir)
                    Open "SALIDA.ARC" For Binary As #3
                    Put #3, , RecFileAll
                    Close #3
                    Label6.Caption = "Descomprimiendo Archivo"
                    Select Case UnpackMethod
                    Case 1
                        UnpackName = "LZ6KUN.COM"
                    Case 2
                        UnpackName = "LZ3KUN.COM"
                    Case 3
                        UnpackName = "COPYUN.COM"
                    End Select
                    vr2 = Shell(UnpackName, vbMinimizedFocus)
                    Do While GetActiveWindow() = hWndAct2
                        vr2 = DoEvents()
                    Loop
                    hWndAct2 = GetForegroundWindow()
                    Do While IsWindow(hWndAct2)
                        vr2 = DoEvents()
                    Loop
                    Label8.Caption="Descomprimido:"+ Str(FileLen(RecFileName))
```

```
                    Kill "SALIDA.ARC"
NoGrabar:
                    Label6.Caption = "Archivo Recibido"
                    State = 1
                    MSComm1.InputLen = 1
                    Kill "ENTRADA.CMP"
                End If
            Case 1      'Awaiting an order
                Rec = MSComm1.Input
                D = Asc(Rec)
                Select Case D
                    Case &H31 'Ping
                        Rec = Chr(&H35)
                        MSComm1.Output = Rec
                        Label1.Caption = "ESTADO: Enlace Establecido"
                    Case &H20 'File Coming, method 1
                        State = 2
                        UnpackMethod = 1
                    Case &H21 'File Coming, method 2
                        State = 2
                        UnpackMethod = 2
                    Case &H22 'File Coming, method 3
                        State = 2
                        UnpackMethod = 3
                    Case &H35 'Pong
                        Label1.Caption = "ESTADO: Enlace Establecido"
                    Case Else
                        'MsgBox "¡¡¡ERRRORRRR DE COMUNICACIÓN!!!"
                        Label1.Caption = "ESTADO: Enlace roto"
                End Select
            Case 2
                Rec = MSComm1.Input
                D = Asc(Rec)
                Longitud = D
                State = 3
            Case 3
                Rec = MSComm1.Input
                D = Asc(Rec)
                Longitud = Longitud + 256 * D
                State = 4
            Case 4
                Rec = MSComm1.Input
                D = Asc(Rec)
                Longitud = Longitud + 65536 * D
                State = 5
            Case 5
                Rec = MSComm1.Input
                D = Asc(Rec)
                Longitud = Longitud + 16777216 * D
                Longitud2 = Longitud
                State = 6
                RecFileName = ""
                Label6.Caption = "Recibiendo Archivo"
                Label7.Caption = "Comprimido: " + Str(Longitud)
                Label8.Caption = ""
            Case 6   'Filling Name of Incoming file
                RecFileName = RecFileName & MSComm1.Input
                If Right(RecFileName, 1) = Chr$(0) Then
                   Open "ENTRADA.CMP" For Output As #1
                   State = 0
                   MSComm1.InputLen = 0
                End If
        End Select
    Case comEvSend
    Case comEvCD
        EVMsg = "Cambio detectado en DCD"
    Case comEvCTS
        'EVMsg = "Cambio detectado en CTS"
    Case comEvDSR
        'EVMsg = "Cambio detectado en DSR"
```

```
    Case comEvRing
        EVMsg = "El teléfono está sonando"
    Case comEvEOF
        EVMsg = "Fin de fichero"
    Case comBreak
        EVMsg = "Interrupción detectada"

    Case comCTSTO
        ERMsg = "Tiempo para CTS sobrepasado"
    Case comDSRTO
        ERMsg = "Tiempo para DSR sobrepasado"
    Case comCDTO
        ERMsg = "Tiempo para DCD sobrepasado"
    Case comFrame
        ERMsg = "Error de trama"
    Case comOverrun
        ERMsg = "Error de sobreescritura"
    Case comRxOver
        ERMsg = "Buffer de recepción lleno"
    Case comRxParity
        ERMsg = "Error de Paridad"
    Case comTxFull
        ERMsg = "Buffer de transmisión lleno"
    Case Else
        ERMsg = "Error desconocido"
End Select

If Len(EVMsg) Then
    Label1.Caption = EVMsg
    EVMsg = ""
ElseIf Len(ERMsg) Then
    Beep
    vr = MsgBox(ERMsg, 1, "Pulse Cancelar para Salir, Aceptar para Ignorar.")
    ERMsg = ""
    If vr = 2 Then
        MSComm1.PortOpen = False
    End If
End If
End Sub
```