UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

Protocolo ARQ de parada y espera

Transmisión de datos
ETSI de Telecomunicación

Himar Alonso Díaz
Lara González Villanueva
Eri Medina Segura
Juan Carlos Molina Rojas

1 Introducción

La presente práctica tiene por objeto la implementación de un *protocolo ARQ de parada y espera*. Dos ordenadores estarán conectados mediante un cable por el *puerto serie* (un *módem nulo*) y se transmitirá a través del mismo, el contenido de un fichero de texto.

El objetivo es lograr una transmisión *correcta* de los datos, por lo que nuestro programa será capaz de detectar errores de *pérdidas de tramas* y errores en los *datos*.

De modo que cuando alguno de estos errores es detectado, se pide al transmisor que envíe de nuevo la misma trama.

En nuestro programa, escrito en `C++`, incluiremos los ficheros *errorP3.h* y *errorP3.obj*, que se encargarán de *simular* estos errores para así comprobar el buen funcionamiento del mismo.

2 Planteamiento por etapas

Esta práctica la hemos desarrollado en varias etapas:

En la primera de ellas desarrollamos la parte de hardware, que consistía en utilizar un cable apantallado de 9 conexiones para conectar dos ordenadores a través del *puerto serie*, mediante un *módem nulo*. Una vez concluido este trabajo práctico, empezamos a diseñar el software que íbamos a necesitar. Inicialmente, tan solo debíamos mandar mensajes, introducidos por teclado, de un ordenador a otro, sin tener en cuenta los posibles errores que pudieran producirse en el canal durante la retransmisión.

En la segunda, en lugar de enviar directamente mensajes introducidos por el teclado, se tomaba un archivo de texto que íbamos enviando por tramas. Según el receptor fuera recibiéndolas, debía pedir la trama siguiente y tener en cuenta que la secuencia en la que se enviaran las tramas fuera la correcta.

A partir de la tercera etapa, empezamos a añadir unos módulos de error, que producían pérdidas en las tramas y cambios en los caracteres de éstas. Para poder detectar estos errores y pedir el reenvío de la trama errónea, transmitíamos una *redundancia* junto con el mensaje.

3 Organigramas

En los siguientes organigramas se tiene una descripción *gráfica* de la secuencia de procesos de los dos programas:

- En la Figura 1 (Página 3) se muestran los procesos que utilizan tanto el transmisor como el receptor, para realizar las configuraciones iniciales.
- En la Figura 2 (Página 4) se muestra el diagrama correspondiente al transmisor.
- En la Figura 3 (Página 5) se muestra el diagrama correspondiente al receptor.

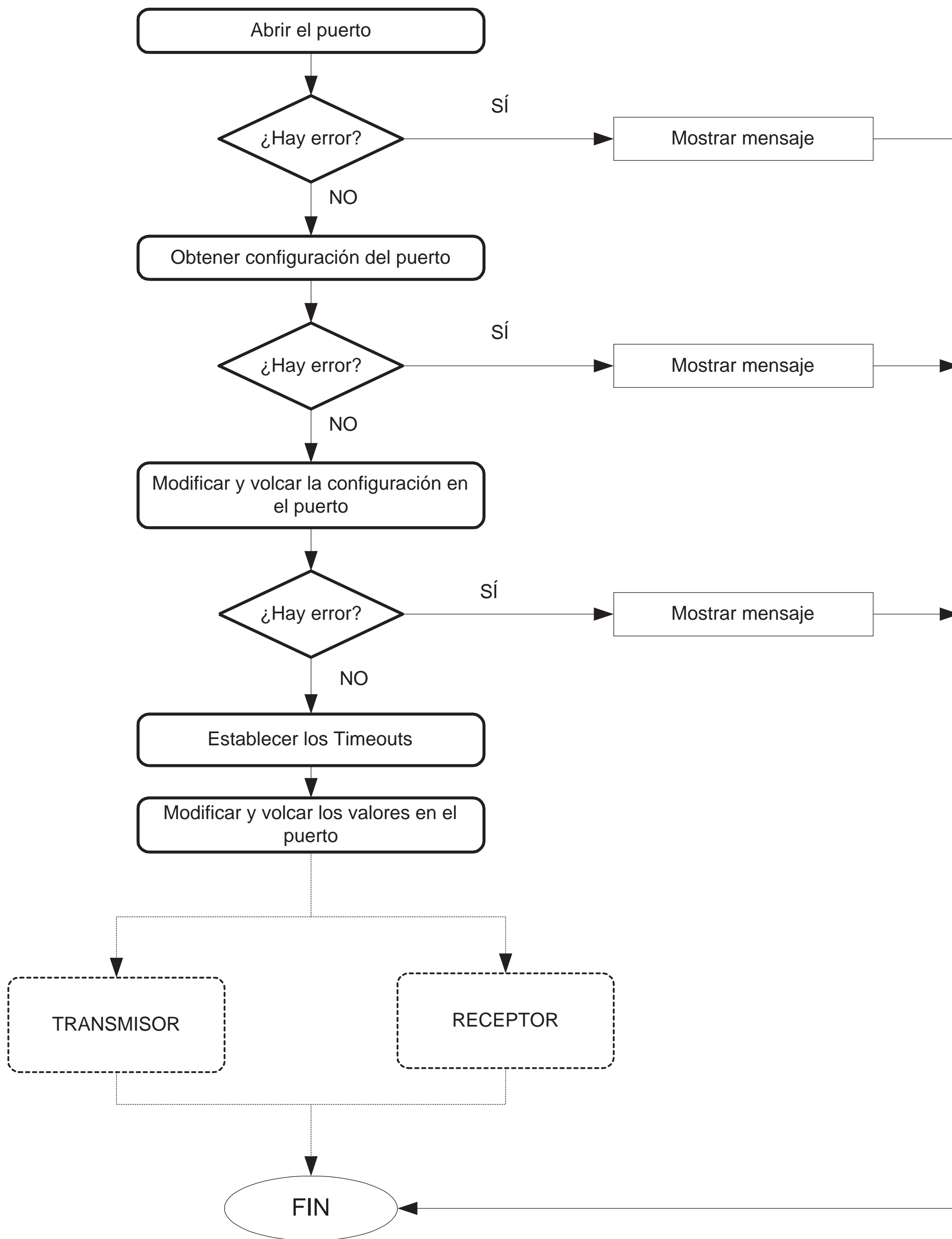

Figura 1: Diagrama de procesos *comunes* al transmisor y al receptor

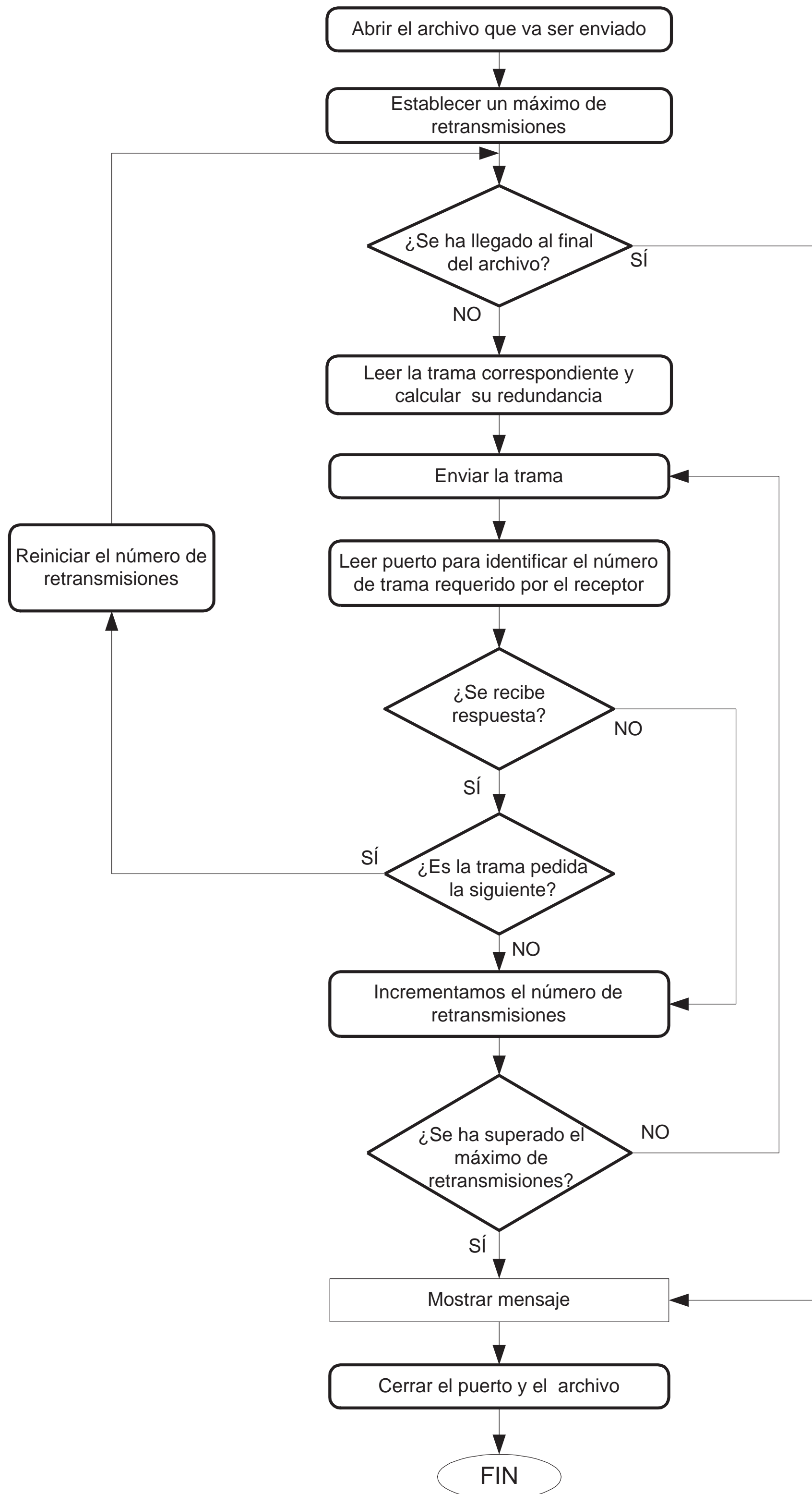

Figura 2: Diagrama del *transmisor*

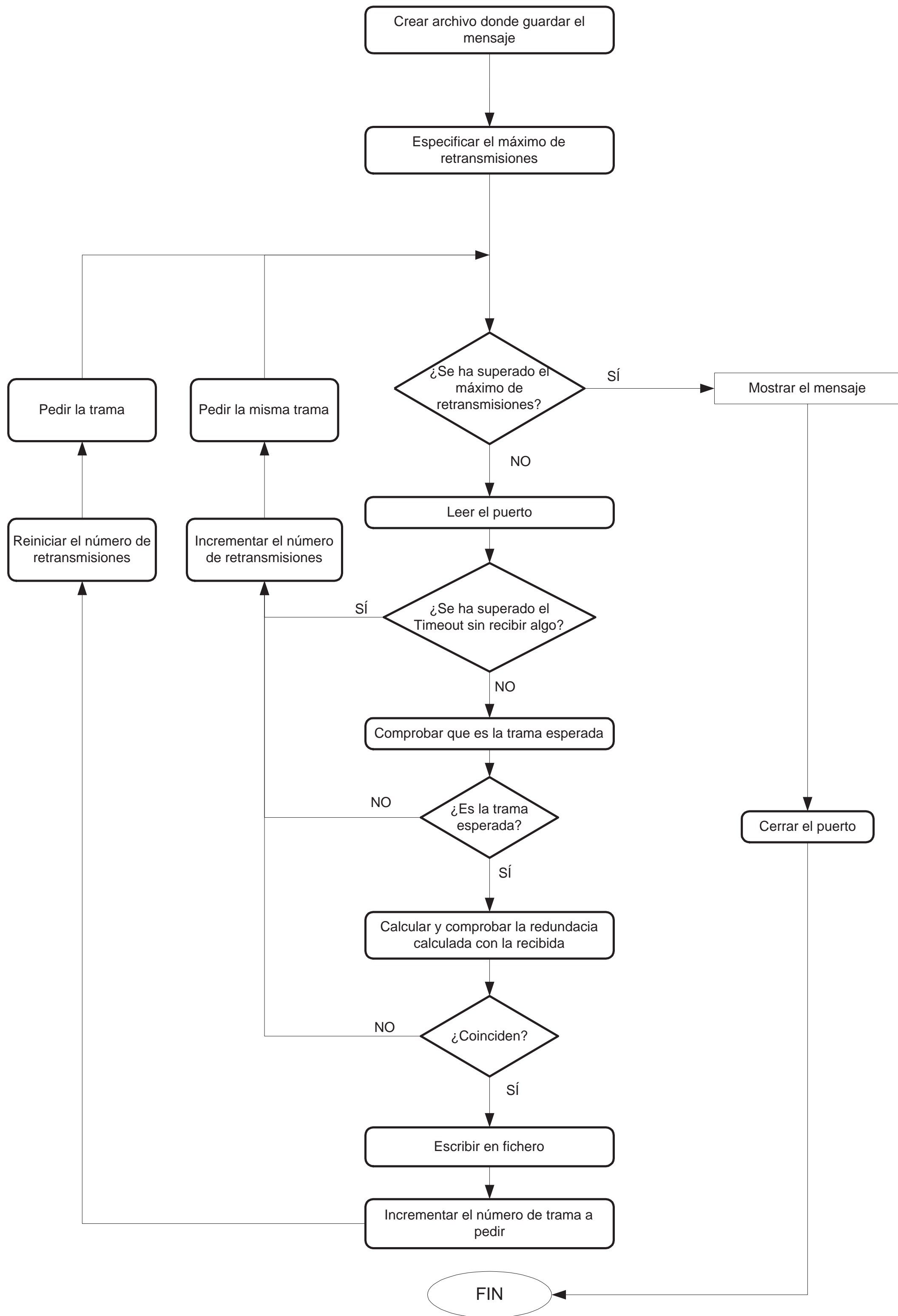

Figura 3: Diagrama del *receptor*

4 Código del transmisor

Este es el *código fuente* del transmisor:

```
//-----------------------------------------------------------------------------------------------


// Declaramos de los paquetes que utilizamos en el programa
#include <conio.h>
#include <iostream.h>
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "errorP3.h"
#include "windows.hpp"

// Declaramos la parte 'no identidad' de la matriz que utilizaremos para el
// cálculo de la redundancia
const int Fila = 11;
const int Columna = 4;

int Matriz[Fila][Columna] = { {1,1,0,0}, {1,0,1,0}, {0,1,1,0}, {1,1,1,0},
                              {1,0,0,1}, {0,1,0,1}, {1,1,0,1}, {0,0,1,1},
                              {1,0,1,1}, {0,1,1,1}, {1,1,1,1} };


//-----------------------------------------------------------------------------------------------


// Función para el cálculo de la redundancia
void Redundancia (char datos[datosxtrama], int Matriz[Fila][Columna],
                  char paridad[])
{
  int paridadtemp [4]={0,0,0,0};
  int i, j;
  for (i=0; i<=3; i++)
  {
    for (j=0; j<=10; j++)
    {
      paridadtemp[i] = paridadtemp[i] + datos[j]*Matriz[j][i];
    }
    paridad[i] = paridadtemp[i]%84;
    if (paridad[i] == 0)
```

```
    {
      paridad[i] = 32;
    }
    else
    {
      paridad[i] = paridad[i] + 39;
    }
  }
}


//-----------------------------------------------------------------------------------------------------


// Programa principal
void main ()
{

// Declaramos las variables que utilizaremos al inicio del programa
HANDLE Puerto;
DCB miDCB;
int correcto;
char salida;

// Intentamos abrir el puerto 1, si no podemos intentamos abrir el puerto 2.
Puerto=CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,
                  0,NULL,OPEN_EXISTING,0,NULL);

if (Puerto == INVALID_HANDLE_VALUE)
{
  Puerto=CreateFile("COM2",GENERIC_READ|GENERIC_WRITE,
                    0,NULL,OPEN_EXISTING,0,NULL);
}

// En caso de que no podamos abrir ninguno de los dos, se nos avisa y termina
// el programa.
if (Puerto == INVALID_HANDLE_VALUE)
{
  cout << "\n\nNo ha podido abrirse el puerto.";
  Sleep(500);
  return;
}

// Si hemos abierto un puerto, obtenemos su configuración.
cout << "\n\nPuerto abierto.  Obteniendo configuracion...";
```

```
Sleep(500);

// 'correcto' nos indica si hemos podido obtener la configuración del puerto.
correcto = GetCommState (Puerto, &miDCB);

// Si no hemos podido, termina el programa.
if (correcto == 0)
{
  cout << "\n\nHa sido imposible capturar la configuracion del puerto.";
  Sleep(500);
  CloseHandle (Puerto);
  return;
}
else
{
  cout << "\n\nSe ha capturado la informacion del puerto.  Configurandola...";
  Sleep(500);
}

// Si hemos obtenido la configuración del puerto, procedemos a modificarla.
miDCB.BaudRate = 9600;
miDCB.ByteSize = 8;
miDCB.Parity = NOPARITY;
miDCB.StopBits = ONESTOPBIT;

// Guardamos la nueva configuración del puerto.
correcto = SetCommState (Puerto, &miDCB);

// Al igual que antes, si 'correcto' nos indica que no hemos podido guardar la
// nueva configuración del puerto, termina el programa.
if (correcto == 0)
{
  cout << "\n\nHa sido imposible modificar la configuración del puerto";
  Sleep(500);
  CloseHandle (Puerto);
  return;
}
else
{
  cout << "\n\nSe ha configurado el puerto correctamente.\n";
  Sleep(500);
}

// Especificamos el tiempo de espera
COMMTIMEOUTS Timeouts;
```

```
GetCommTimeouts (Puerto, &Timeouts);
Timeouts.ReadTotalTimeoutConstant = 10000;
SetCommTimeouts (Puerto, &Timeouts);

// Abrimos el archivo que queremos enviar
FILE* archivo;
archivo = fopen("mensaje.txt", "r");

// Inicializamos las variables del tipo trama_datos declarado en errorP3.h
trama_datos trama;
trama.numeracion = 0;
trama.paridad [4] = 0,0,0,0;

// Variables que necesitaremos para el envío del mensaje.
DWORD Longitud;
int nueva_secuencia = 0;
int contador = 2;
int guay;

// Función necesaria para el correcto funcionamiento del módulo de error.
randomize();

// Leemos la primera trama.
fgets (trama.datos, datosxtrama+1, archivo);

// Calculamos la redundancia para esta primera trama.
Redundancia(trama.datos, Matriz, trama.paridad);

// Se avisa al usuario de que se va a proceder al envío del archivo.
cout << "\n\nEnviando archivo...\n";
Sleep(500);

// Enviamos la primera trama.
guay = Envio(trama, Puerto);

// Esperamos la respuesta del receptor.
ReadFile (Puerto, &nueva_secuencia, sizeof(int), &Longitud, NULL);

// Si no recibimos respuesta, le mandamos de nuevo la misma trama.
if (Longitud == 0)
{
  nueva_secuencia = trama.numeracion;
}

// Especificamos un máximo de cinco retransmisiones de una misma trama.
```

```
while ((contador<=5))
{
  // Mientras no lleguemos al final del archivo...
  if (!feof(archivo))
  {
    // Si piden la trama siguiente a la enviada...
    if (nueva_secuencia == (trama.numeracion + 1)%8)
    {
      // Leemos una nueva trama e inicializamos el contador de trama.
      trama.numeracion = (trama.numeracion +1)%8;
      if (fgets (trama.datos, datosxtrama+1, archivo)!=NULL)
      {
        contador = 1;
      }
      else
      {
        break;
      }
      // Calculamos la redundancia de la nueva trama.
      Redundancia(trama.datos, Matriz, trama.paridad);
    }
    // Si vuelven a pedir la misma trama incrementamos el contador,
    // ya que hay una nueva retransmisión de una misma trama.
    else
    {
      contador++;
    }
    // Enviamos la trama correspondiente.
    guay = Envio(trama, Puerto);
    // Esperamos la respuesta del receptor.
    ReadFile (Puerto, &nueva_secuencia, sizeof(int), &Longitud, NULL);
    // Si no hay respuesta, le reenviamos la trama correspondiente.
    if (Longitud == 0)
    {
      nueva_secuencia = trama.numeracion;
    }
  }
  // Cuando hemos llegado al final del archivo...
  else
  {
    // Si piden la última trama...
    if (nueva_secuencia == trama.numeracion)
    {
      // El contador se incrementa y volvemos a reenviarla.
      contador++;
```

```
      guay = Envio(trama, Puerto);
      ReadFile (Puerto, &nueva_secuencia, sizeof(int), &Longitud, NULL);
      if (Longitud == 0)
      {
        nueva_secuencia = trama.numeracion;
      }
    }
    // Si piden una trama nueva, como ya hemos terminado de mandar
    // el archivo, el programa termina. Ellos, en este caso,
    // realizarán cinco intentos de escucha, y como no recibirán
    // nada, a continuación, esperarán a que salte el 'Timeouts'.
    else
    {
      break;
    }
  }
}

// El programa se despide cordialmente del usuario.
cout << "\n\nHASTA PRONTO :)";
Sleep(3000);

// Se cierran el archivo y el puerto.
fclose (archivo);
CloseHandle (Puerto);

// Termina el programa.
}


//-----------------------------------------------------------------------------------------------
```

5 Código del receptor

Este es el *código fuente* del receptor:

```
//--------------------------------------------------------------------------------------------------


// Declaración de los paquetes que utilizamos en el programa.
#include <conio.h>
#include <iostream.h>
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include "errorP3.h"
#include "windows.hpp"

// Declaramos la parte 'no identidad' de la matriz que utilizaremos para el
// cálculo de la redundancia, y sus variables asociadas.
const int Fila = 11;
const int Columna = 4;

int Matriz[Fila][Columna] = { {1,1,0,0}, {1,0,1,0}, {0,1,1,0}, {1,1,1,0},
                              {1,0,0,1}, {0,1,0,1}, {1,1,0,1}, {0,0,1,1},
                              {1,0,1,1}, {0,1,1,1}, {1,1,1,1} };

// Variables asociadas a la función 'Redundancia'.
bool paridad_igual;
char paridad_calculada [redundanciaxtrama];

// Función para el cálculo de la redundancia.
void Redundancia (char datos[datosxtrama], int Matriz[Fila][Columna],
                  char paridad[])
{
  int paridadtemp [4]={0,0,0,0};
  int i, j;
  paridad_calculada [4]= 0,0,0,0;
  for (i=0; i<=3; i++)
  {
    for (j=0; j<=10; j++)
    {
      paridadtemp[i] = paridadtemp[i] + datos[j]*Matriz[j][i];
    }
    paridad[i] = paridadtemp[i]%84;
```

```
    if (paridad[i] == 0)
    {
      paridad[i] = 32;
    }
    else
    {
      paridad[i] = paridad[i] + 39;
    }
  }
}


//-----------------------------------------------------------------------------------------------


// Programa principal
void main ()
{

// Declaramos variables generales que utilizaremos en el programa.
HANDLE Puerto;
DCB miDCB;
int correcto;

// Intentamos abrir el puerto 1, si no podemos, intentamos abrir el puerto 2.
Puerto = CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,
                    0, NULL, OPEN_EXISTING, 0, NULL);

if (Puerto == INVALID_HANDLE_VALUE)
{
  Puerto = CreateFile("COM2",GENERIC_READ|GENERIC_WRITE,
                      0, NULL, OPEN_EXISTING, 0, NULL);
}

// En caso de que no podamos abrir ninguno de los dos, se nos avisa y termina
// el programa.
if (Puerto == INVALID_HANDLE_VALUE)
{
  cout << "\nNo ha podido abrirse el puerto.";
  CloseHandle (Puerto);
  return;
}

// Si hemos abierto un puerto, obtenemos su configuración.
cout << "\n\nPuerto abierto.  Obteniendo configuracion...";
```

```
Sleep(500);

// 'correcto' nos indica si hemos podido obtener la configuración del puerto.
correcto = GetCommState (Puerto, &miDCB);

// Si no hemos podido, termina el programa.
if (correcto == 0)
{
  cout << "\n\nHa sido imposible capturar la configuracion del puerto.";
  Sleep(500);
  CloseHandle (Puerto);
  return;
}
else
{
  cout << "\n\nSe ha capturado la informacion del puerto.  Configurandola...";
  Sleep(500);
}

// Si hemos obtenido la configuración del puerto, procedemos a modificarla.
miDCB.BaudRate = 9600;
miDCB.ByteSize = 8;
miDCB.Parity = NOPARITY;
miDCB.StopBits = ONESTOPBIT;

// Guardamos la nueva configuración del puerto.
correcto = SetCommState (Puerto, &miDCB);

// Al igual que antes, si 'correcto' nos indica que no hemos podido
// guardar la nueva configuración del puerto, termina el programa.
if (correcto == 0)
{
  cout << "\n\nHa sido imposible modificar la configuración del puerto";
  Sleep(500);
  CloseHandle (Puerto);
  return;
}
else
{
  cout << "\n\nSe ha configurado el puerto correctamente.\n\n";
  Sleep(500);
}

// Especificamos el tiempo de espera.
COMMTIMEOUTS Timeouts;
```

```
GetCommTimeouts (Puerto, &Timeouts);
Timeouts.ReadTotalTimeoutConstant = 10000;
SetCommTimeouts (Puerto, &Timeouts);

// Creamos el archivo en el que guardaremos el mensaje.
FILE* archivo;
archivo = fopen("mensaje.txt", "w");

// Establecemos las variables que necesitaremos para el control de las tramas.
trama_datos trama;
int ACK = 0;

// Variables que necesitaremos para la recepción del mensaje.
DWORD Longitud;
bool guay;
int max_retransmisiones = 0;

// Función necesaria para el correcto funcionamiento del módulo de error.
randomize();

// Se avisa al usuario de que se va a proceder al envío del archivo.
cout << "\nRecibiendo archivo...\n";
cout << "\n";
Sleep(500);

// Especificamos un máximo de cinco retransmisiones de una misma trama.
while (max_retransmisiones <= 5)
{
  // Escuchamos el puerto.
  ReadFile (Puerto, &trama, sizeof(trama_datos), &Longitud, NULL);
  // Si no recibimos nada, pedimos de nuevo la misma trama
  // y aumenta el contador.
  if (Longitud == 0)
  {
    guay = Envio (ACK, Puerto);
    max_retransmisiones++;
  }
  // Si recibimos una trama...
  else
  {
    // Comprobamos que es la solicitada, en caso de que no lo sea,
    // la pedimos de nuevo, y se incrementa el contador.
    if (trama.numeracion != ACK)
    {
      guay = Envio (ACK, Puerto);
```

```
        max_retransmisiones++;
      }
      // Si recibimos la trama pedida, calculamos su redundancia.
      else
      {
        Redundancia (trama.datos, Matriz, paridad_calculada);
        // Comprobamos que la redundancia que hemos calculado sea
        // la misma que la que hemos recibido.
        int x;
        paridad_igual = true;
        for (x=0; x<=3; x++)
        {
          paridad_igual = paridad_igual && trama.paridad [x] ==
                          paridad_calculada [x];
        }
        // Si las dos redundancias no coinciden, quiere decir que
        // ha habido un error en el contenido de la trama,
        // por lo tanto, pedimos de nuevo la misma trama.
        if (paridad_igual == false)
        {
          guay = Envio (ACK, Puerto);
          max_retransmisiones++;
        }
        // Si las dos redundancias son idénticas, escribimos la trama
        // en el archivo, y pedimos la siguiente trama.
        else
        {
          cout << trama.datos;
          max_retransmisiones = 0;
          ACK++;
          ACK = ACK%8;
          fputs(trama.datos, archivo);
          guay = Envio (ACK, Puerto);
        }
      }
    }
  }

  // Cuando se supera el número máximo de retransmisiones, finaliza el programa.
  if (max_retransmisiones > 5)
  {
    cout << "\nSe ha superado el número máximo de retransmisiones.  Cerrando...";
    Sleep(3000);

    // Se cierran el archivo y el puerto.
```

```
  CloseHandle (Puerto);
  fclose (archivo);

  // Termina el programa.
  return;
}
}


//-------------------------------------------------------------------------------------------------
```