PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 3/14 — A1

(11) International Publication Number: **WO 99/26130**

(43) International Publication Date: 27 May 1999 (27.05.99)

(21) International Application Number: PCT/US98/24342

(22) International Filing Date: 13 November 1998 (13.11.98)

(30) Priority Data:
08/970,709 14 November 1997 (14.11.97) US

(71) Applicant: E–PARCEL, LLC [US/US]; Suite 300, 29 Crafts Street, Newton, MA 02158 (US).

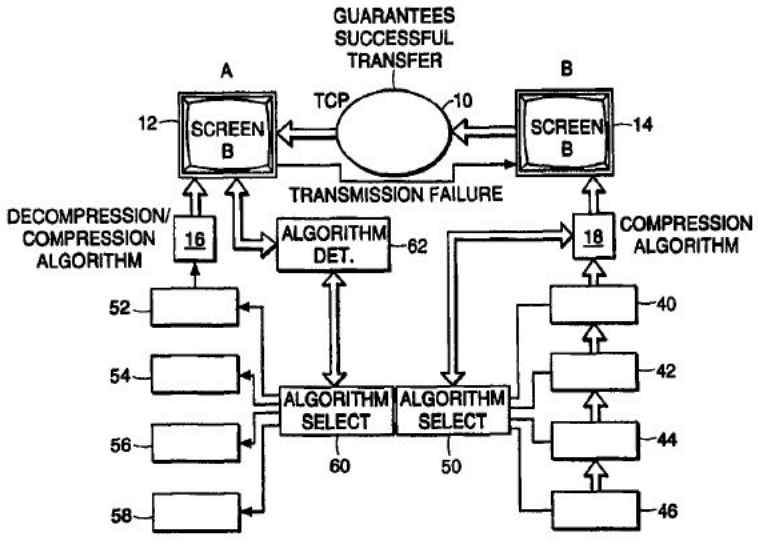
(72) Inventors: KOBATA, Hiroshi; Apartment 201, 50 Watertown Street, Watertown, MA 02172 (US). GAGNE, Robert, A.; 995 Southern Artery, Quincy, MA 02169 (US). TONCHEV, Theodore, C.; 162 5th Street, Cambridge, MA 02141 (US).

(74) Agent: TURANO, Thomas, A.; Testa, Hurwitz & Thibeault, LLP, High Street Tower, 125 High Street, Boston, MA 02110 (US).

(81) Designated States: AU, CA, CN, IL, JP, KR, NO, NZ, SG, European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: REMOTE CONTROL SYSTEM WHICH MINIMIZES SCREEN REFRESH TIME BY SELECTING COMPRESSION ALGORITHM



(57) Abstract

A system is provided for the remote control of one computer from another in which selectable compression speeds are utilized to minimize overall screen refresh time. In one embodiment, an algorithm selection module at one computer chooses the highest compression available corresponding to a worst case scenario, followed by measurement of the compression time and the transmission time, with the ratio of compression time to transmission time being used to select a decreased compression, thereby to lower compression time and consequently lower the overall screen refresh time. By adjusting both the send time and the compression time on the transmit side, the above ratio can be made to equal one, which corresponds to the most efficient utilization of the available bandwidth and CPU power, which in turn translates into the quickest screen refresh time for the remote control operation.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

REMOTE CONTROL SYSTEM WHICH MINIMIZES SCREEN REFRESH TIME BY SELECTING COMPRESSION ALGORITHM

FIELD OF INVENTION

This invention relates to the remote control of computers and more particularly to an emulation system in which screen refresh times associated with one computer controlling another computer are minimized taking into account the demographics of the network and the sending machine.

BACKGROUND OF THE INVENTION

One of the main problems for a consumer is oftentimes incompatibility of his equipment and the programs sought to be run. For instance, an individual may have a printer that is incompatible with the driver loaded to operate it. Typically, the user calls an administrator who telephonically instructs the individual as to how to program his computer in order to make the printer work. This may involve downloading printer drivers or other software in a time consuming process in which the administrator must ask the user questions. The user, on the other hand, may be insufficiently knowledgeable to answer the administrator's questions, resulting in frustration on both the user's part and that of the administrator.

In an effort to minimize such frustrations, systems, referred to herein as emulation systems, have been developed so that the administrator can "see" what is happening at the user's terminal. In order to do this in the past remote control programs have been developed which capture the low level graphic calls and send them to the administrator's computer for display. At the administrator's side these low level graphic calls are utilized to provide the screen at the administrator's side with an exact duplicate of the user's screen. Such a system is commercially available as model pcANYWHERE from Symantec of Cupertino, California. In this system the

administrator can understand what is happening at the user's side and verbally instruct the user what to do.

One of the problems with prior emulation systems is that the screen refresh at the administrator's side is slow, or in general not optimized to the demographics of the network and the users machine. For instance, if the user has a relatively slow modem connected to the network, but is utilizing a compression algorithm that emphasizes compression speed over efficiency, this would have a negative impact on the screen refresh rate. The result for the administrator is that the administrator would select or click on an icon and have to wait an inordinate amount of time for a response. The reason for such delays has to do with the demographics of the network and the two machines in that inappropriate compression algorithms are chosen.

In an effort to speed up the response of such systems, various techniques have been utilized. One of these techniques involves the use of low level drivers to capture graphics calls. However, these drivers can make the system unstable and require much disk and RAM memory space. As a result, oftentimes the screen refresh is often corrupted when moving graphics are encountered and has a stilted appearance.

The artifacts are in general caused by the order in which the graphic calls are trapped, and the different levels of graphics calls which are available. These systems are also inferior in displaying bitmapped graphics and moving images. The largest problem with the above systems is that they can take as much as 10 megabytes of disk space on both sides, and require a reboot after installation before using.

In the prior systems there is another problem in that by relying on the graphics calls the images are taken in parts and are displayed on the administrator's side in unnatural fashion due to the arbitrary order in which the graphics calls are trapped. These systems have to wait for other

applications to make graphic calls in order to know what to update and rely on the applications to make graphics calls known to the systems. If an application performs a graphics operation not known to the system, that information is not transferred over to the administrator.

As a result, the image the administrator sees is incorrect, since the entire system is dependent on other applications to perform known operations with the operating system. For applications that perform operations that are unknown, the system ignores what could potentially be the problematic area.

Thus, for the newer peripherals coupled to the user's computer, in the past the problem could be ignored since the administrators system could not see it.

SUMMARY OF THE INVENTION

In order to speed up the screen refresh potion of the system, low level drivers trapping graphics calls are eliminated in favor of increasing speed through the alteration of the compression and decompression algorithms used at the user and administrator sides to choose the appropriate ratio of transmission time and compression time which matches the demographics of the network and that of the user's computer. In one embodiment, the speed of the refresh at the administrator's side is increased to the maximum extent possible to eliminate delays in presenting the results of an operation on the user's computer.

For instance, assuming that the user's computer has a CPU speed of 200 mhz and further assuming a local area network with a mean transmission rate of ~800 kilobytes per second, best compression algorithm would provide a full screen refresh in less than 0.7 seconds including compression and transmission. If an incorrect algorithm for modem speed were chosen at the user's side, such as 28 kilobytes per second, then the refresh time at the administrator's side would be 2.5 seconds, clearly 300% longer than that achievable if the appropriate compression rate were chosen.

In the above case it would be desirable to choose the lowest compression rate, eq. one that emphasizes compression speed over resulting size. In this case, e.g., for a local area network, a compression rate of 13% would yield the above 0.7 second refresh time, given a transmission speed of ~800kb.

In the subject invention there are four different compression algorithms from which to choose. The first is the lowest compression rate algorithm comprising a run length encoding algorithm. This algorithm converts a run of the same byte with a count of that byte. A second algorithm selectable by the subject system for the next higher compression rate is a Huffman compression algorithm preceded by run length encoding, or RLE. A third algorithm selectable by the subject system for the next higher compression rate is a modified adaptive Huffman compression algorithm using a 9 bit tree entry size, again preceded by RLE. Finally, a fourth compression algorithm is identical to the above but with 13 bit tree entry sixes, for the highest compression rate.

The first algorithm is characterized by it's speed. Typically, this algorithm will compress a megabyte in less than 0.2 seconds. This algorithm is to be selected when network bandwidth is not a problem.

The second algorithm is also characterized by it's speed, but with more resulting data size efficiency, and is to be used in high speed networks with heavy traffic.

The third algorithm is characterized by its data size efficiency, in which a megabyte is compressed down to 4 kilobytes. This algorithm is useful for internet applications because of the internet's heavy traffic and latency.

The fourth algorithm is characterized by its extreme data size efficiency in that it can compress a megabyte of screen data down to approximately 2 kilobytes or less. However, the last two algorithms are relatively slow, eg. 3 seconds vs. 0.2 seconds.

While the above describes four different compression algorithms, other compression/decompression algorithms are within the scope of this invention. Regardless, the compression algorithm selected is based on the rate of send time to compression time, with the selection seeking to cause this ratio to equal one. In order to select the decompression algorithm at the user's side, the transmission from the user to the administrator is monitored and the transmission time is measured for every single refresh. In one embodiment, this involves measuring the start of the send and end of the send for a TCP network transmission. Likewise, the actual compression is measured for every single refresh in which the CPU tick count is taken at the start of the compression and at the end of the compression; and the end of the compression; and the difference is used to calculate the compression speed in milliseconds.

In one embodiment, the screen is divided into grids and a checksum recorded for each grid. This check sum is compared to the previous check sum, and when it differs, the grid is marked as "dirty". Once the entire screen has been checked, all the dirty grids are collected and compressed. This is one refresh. This process is done on a grid by grid basis until the whole screen has been checked.

The selection process operates as follows. After an initial measurement has been made of compression time and transmission time, the ratio is calculated, and if less than one, the algorithm having a lower compression rate is selected, thereby decreasing compression time.

In one embodiment, the first algorithm selected is the one having the highest compression rate, assuming the worst case scenario. Thereafter, through an iterative process the selection settles on the optimal algorithm given the demographics of the network and the user's computer.

The compression time and the transmission time is averaged out from refresh to refresh so that the ratio reflects the overall demographics. The algorithm chosen is that for which the ratio is as close to one as possible.

A system is provided for the remote control of one computer from another in which selectable compression speeds are utilized to minimize overall screen refresh time. In one embodiment, an algorithm selection module at one computer chooses the highest compression available, corresponding to a worst case scenario followed by measurement of the compression time and the transmission time, with the ratio of compression time to transmission time being used to decrease compression, thereby to lower compression time and consequently lower the overall screen refresh time. By adjusting both the send time and the compression time on the transmit side, the above ratio can be made to be equal to one, which corresponds to the most efficient utilization of the available bandwidth and CPU power, which in turn translates into the quickest screen refresh time for the remote control operation.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features of the Subject Invention will be better understood with reference to the Detailed Description taken in conjunction with the Drawings, of which:

Figure 1 is block diagram of two networked computers, with one computer corresponding to that of the user, and the other computer corresponding to that of the administrator;

Figure 2 is a block diagram of the computers of Figure 1 illustrating the transmission of the information on the screen of the user's computer to the screen of the administrator's computer;

Figure 3 is a block diagram of the transmission of modifications specified at the administrator's screen to that of the user's screen indicating the control of the user's computer by the administrator's computer;

Figure 4 is a block diagram of the subject system in which the compression -- algorithm to be used by the user's computer is automatically selected based on the demographics

of the network and the user's computer, which selection is transmitted to the administrator's computer for use in the corresponding compression algorithms therein;

Figure 5 is a table illustrating the header portion for the data transmitted for a single screen cycle indicating identification of the compression algorithm used;

Figure 6 is a flowchart illustrating the derivation of a ratio of send time to compression time used in the algorithm selection module of figure 4; and,

Figure 7 is a graph illustrating a comparison of compression algorithms in terms of transfer speed vs. compression speed.

DETAILED DESCRIPTION

Referring now to Figure 1 in a typical networked situation a network 10 is used to connect a terminal A, herein referred to as terminal 12, to terminal B, herein referred to as terminal 14.

Each of these terminals has a CPU coupled, thereto referred to as 16 and 18, with keyboards 20 and 22 coupled to respective CPUs. Mice 24 and 26 are coupled to respective CPUs 16 and 18 as illustrated.

Referring now to Figure 2, in an emulation system, terminal 14 transmits its screen 30 over network 10 to terminal 12, such that its screen is duplicated as 30' at terminal 12. It is the purpose of this transfer to alert the administrator to exactly what is displayed at the user's terminal so that corrective action can be taken by the user either through verbal instructions given to the user by the administrator or, as illustrated in Figure 3, by the transmission of modifications 34 from terminal 12 to terminal 14 over network 10.

As mentioned hereinbefore, it is important that there be minimal delay between actions taken by the administrator via keyboard 20 or mouse 24 and a resulting operation on terminal 14 which change is immediately displayed on terminal 12. The ability to quickly display

operations and results on the administrator's terminal significantly reduces administrator frustration and fatigue, while at the same time providing more efficient transfer of information to the user or more particularly to the user's terminal. Regardless of whether or not information is verbally communicated to the user or is automatically downloaded to the users terminal it is important that the administrator's screen be refreshed in the shortest possible time.

Factors which affect the screen refresh are the characteristics of the network, herein referred to as demographics, which includes bandwidth, transmission speed, traffic and other factors. Likewise, the screen refresh rate at the administrator's side is intimately connected with the demographics of the user's computer, namely CPU speed, modem speed, available memory, and other factors.

Referring to Figure 4, as can be seen, each of the CPUs 16 and 18 is provided with a selectable number of compression and decompression algorithms, with the compression algorithms at the user's side bearing reference characters 40, 42, 44 and 46. These algorithms, in one embodiment, are ordered from the highest to the lowest compression, the purpose of which will be hereinafter. The algorithm selected at the user's side is accomplished through the utilization of an algorithm selection module, 50, which is coupled not only to CPU 18, but also to each of the algorithms 40-46.

Likewise, at the administrator's side corresponding decompression algorithms 52, 54, 56 and 58, are coupled with CPU 16 in accordance with algorithm selection module 60, upon detection at 62 of the compression algorithm carried by the transmitted data over network 10.

In operation, each screen refresh cycle is provided with a header containing the type of compression algorithm selected at the users side by module 50. The data format of the transmitted data is illustrated in Figure 5 to indicate that as part of the header information the

algorithm selected is identified, with the remainder of the data being that associated with a screen refresh cycle.

This header is detected at 62 and the appropriate algorithm is introduced to CPU 16 for appropriate decompression.

Referring to Figure 6, in one embodiment, algorithm selection module 50 initially chooses for transmission that algorithm which possesses the highest compression. The reason this is done is to be able to accommodate the worst case scenario in which the transmission speed is limited to that of the slowest modem reasonably calculated to be included at the receiver's CPU. Algorithm select module 50 then computes how long it takes to compress a refresh cycle and how long it takes to send this refresh cycle. It does this each and every refresh cycle to permit a ratio of send time to compression time to be calculated for each refresh cycle.

If the ration is equal to 1, then this highest compression algorithm is permitted to continue. If the ratio R of send time to compression time is less than 1 then the algorithm having the next lower compression is selected, which lowers the compression time while at the same time increasing the send time. Measurements are again taken for the next cycle and the ratio recomputed. This iterative process finally settles upon an algorithm which optimally minimizes screen refresh time at the administrator's side.

It will be noted that, as mentioned above, the compression algorithms are ordered according to compression so that the iterative process can settle upon the optimal algorithm.

Referring now to Figure 7, what is shown is a comparison of tradeoffs in compression speed and transfer speed for four different compression algorithms having a descending value of compression from the highest to the lowest. As can be seen the highest transfer speed is for a local area network at about 800 kilobytes per second, followed by a wide area network at about

500 kilobytes per second, followed by the internet at about 80 kilobytes per second, and finally followed by modems which operate at about 10 kilobytes per second.

As can be seen, given the lowest compression algorithm, the transfer time is off the chart for a modem and barely usable for the internet, while being satisfactory both for wide area networks and for local area networks, since the transfer time is under 1.0 seconds. On the other hand, for the highest compression algorithm, it can be seen that the overall transfer time varies little such that few improvements can be made in terms of the type of network over which the data is transmitted. For algorithms having intermediate compression rates, while the chart indicates modest improvement in certain instances, significant improvement in refresh times can nonetheless be effectuated in certain circumstances. It will be appreciated that the graph shows averages, and that certain screen data compresses better with one or the other middle algorithms. The dynamic algorithm switching ensures that the most appropriate algorithm is always chosen.

What is presented below is a program listing in C++ which describes the operation not only of the automatic selection module but also of the operation of the entire system in the selection of optimal compression and decompression algorithms based on system demographics.

APPENDIX

```
// Handshake information
#define REJECT     0x0000
#define VERSION        0x0106

#define WAIT_TIME 15000

// Input codes
#define INPUT_MOUSE                     0x01
#define INPUT_KEYBOARD                  0x02
#define INPUT_DOUBLE_CLICK_MOUSE   0x03
#define INPUT_REFRESH                   0x04
#define INPUT_CLOSE_CONNECTION          0x05
#define INPUT_HOTKEY                    0x06
#define INPUT_PAINTING_PAUSE       0x07
#define INPUT_PAINTING_RESUME      0x08

// hot keys
#define HOTKEY_ALTTAB                   0x0010
#define HOTKEY_CTRLESC                  0x0011
#define HOTKEY_CTRLALTDEL               0x0012

// video codes
#define VIDEO_PAINT                     0x01
#define VIDEO_NO_PAINT                  0x02
#define VIDEO_CLOSE_CONNECTION          0x03
#define VIDEO_PAUSE                     0x04

#define CONNECTION_TRANSFER             0x00
#define CONNECTION_PAUSE           0x01


#define MONITOR_EXIT_SUCCESS            0
#define VIDEO_EXIT_SUCCESS                   1
#define INPUT_EXIT_SUCCESS                   2
#define VIDEO_EXIT_HANDSHAKE_ERROR      3
#define VIDEO_EXIT_HARDWARE_ERROR       4
#define VIDEO_EXIT_DIRECT_DRAW_ERROR    5
#define INPUT_EXIT_HANDSHAKE_ERROR     6
#define VIDEO_EXIT_CLIENT_DOESNT_SUPPORT 7
#define VIDEO_EXIT_HANG                      8

// Max mouse movement for mouse_event
#define MOUSE_X 0xffff
#define MOUSE_Y 0xffff
```

```
#define PADDING_DIVISOR 401

// Misc Defines
#define DIB_HEADER_MARKER   ((WORD) ('M' << 8) | 'B')
#define BITS_BYTE   8


// these are for changing the cursors to
// windows defaults
// these are not in any headers
#define OCR_NORMAL          32512
#define OCR_IBEAM           32513
#define OCR_WAIT            32514
#define OCR_CROSS           32515
#define OCR_UP              32516
#define OCR_SIZENWSE        32642
#define OCR_SIZENESW        32643
#define OCR_SIZEWE          32644
#define OCR_SIZENS          32645
#define OCR_SIZEALL         32646
#define OCR_NO              32648
#define OCR_APPSTARTING     32650


#define AGENT_UI_WAITING      _T("Waiting for Administrator
to connect.")
#define AGENT_UI_CONNECTING        _T("Administrator
reuqesting connect...")
#define AGENT_UI_CONNECTED         _T("Administrator
connected.")


//////////////////////////////////////
// Custom windows messages

#define USER_EXITLOOP   WM_USER+1
#define USER_PAUSE       WM_USER+2
#define USER_RESUME      WM_USER+3

//////////////////////////////////////
// Grid defines

#define GRID_HEIGHT      12 //12 //6
#define GRID_WIDTH       16 //16 //8
#define GRID_COUNT (GRID_WIDTH*GRID_HEIGHT)
#define OFFSCREEN_WIDTH 8 //8  //1
```

```
#define PADDING 8 //8 //4



////////////////////////////////////////////////////
// Error codes

#define CONNECT_SUCCESS                         0x01
#define CONNECT_INCORRECT_VERSION       0x02
#define CONNECT_NOT_AVAILABLE           0x03
#define CONNECT_AGENT_REJECT            0x04
#define CONNECT_HARDWARE_INCOMPATIBLE   0x05
#define CONNECT_CLIENT_INCOMPATIBLE         0x06
#define CONNECT_VIDEO_HANG                  0x07

const int STATIC_BUFFER = 256;

// structures for sending a keyboard or mouse event over
the connection
// implementation notes: these contain for information then
needed for
// a minimal inplementation
// they are the parameters from keydb_event and mouse_event


// Description of the protocol
// Video Loop
// Input Loop

struct KeyboardEvent
{
     BYTE Vk;
     // ** fields that are part of the func **//
     BYTE Scan;
     DWORD dwFlags;
     DWORD dwExtraInfo;
     DWORD dwRepeat;
};


struct OtherEvent
{
     int HotKeyId;
};

struct MouseEvent
```

```
{
     DWORD dwFlags;
     DWORD dx;
     DWORD dy;
     // ** fields that are part of the func **//
     DWORD dwData;
     DWORD dwExtraInfo;
     DWORD dwRepeat;
};


struct InfoBlock
{
     long cbCompressedSize;
     long cbFullSize;
     long nDirtyCount;
     DWORD fCompression;
     DWORD fStatus;
     DWORD fCommands;

     // utilities
     InfoBlock ( )
     {
          Clear ( );
     }
     void Clear ( )
     {
          cbCompressedSize = 0;
          cbFullSize       = 0;
          nDirtyCount      = 0;
          fStatus          = 0;
          fCommands = 0;
     }
     enum { PALETTE_AVAIL = 0x01 };
};


struct Status
{
     Status ( )
     {
          fStatus = 0;
     }
     void SetPause ( )
     {
          fStatus |= PAUSE;
```

```
    }
    void SetRefresh ( )
    {
        fStatus |= REFRESH;
    }
    bool Refresh ( )
    {
        if ( fStatus & REFRESH )
            return true;
        else return false;
    }
    bool Pause ( )
    {
        if ( fStatus & PAUSE )
            return true;
        else return false;
    }
    void Clear ( )
    {
        fStatus = 0;
    }
    DWORD fStatus;
    enum { PAUSE = 0x02, REFRESH = 0x04 };
};

struct DirtyBlock
{
    short xPos;
    short yPos;

    // utilities
    void Mark ( int x, int y )
    {
        xPos = (short) x;
        yPos = (short) y;
    }
};

struct HardwareInfo
{
    long ScreenWidth;
    long ScreenHeight;
    long MaxGridCount;
    long ByteCount;
    bool bFail;

    HardwareInfo ( )
```

```
    {
        bFail = false;
    }
    void SetFail ( )
    {
        bFail = true;
    }
    bool GetFail ( ) { return bFail; }
};

// Global Utilities

//void ResourceMessageBox (UINT, UINT=IDS_CAPTION,
DWORD=MB_OK);

// clientvideo.h

// July 30, 1997
// Rob Gagne

// Purpose: Does the job of working with the display in the
form of
// a Video object, calcs the checksum, builds the dirty
buffer and
// compresses it.

class ClientVideo
{
public:
    ClientVideo ( );
    ~ClientVideo ( );

    bool OpenSession ( HWND );
    void CloseSession ( );
    bool ProcessFrame ( InfoBlock&, DirtyBlock*, const
LPBYTE, DWORD );

    long TotalBufferSize ( ) { return m_cbTotalBufferSize;
}
    void QueryHardware ( HardwareInfo& );
    long GridCount ( )  { return GRID_WIDTH * GRID_HEIGHT;
}
    int  MaxPalSize ( ) { return m_display.MaxPalSize ( );
}
    bool GetPalette ( InfoBlock&, LPPALETTEENTRY );

    // Process iteration commands
```

```
        enum { FORCE_PAINT = 0x01 };
private:
        bool ProcessIteration ( InfoBlock&, DirtyBlock*, DWORD
);
        bool ProcessIterationNoLock ( InfoBlock&, DirtyBlock*,
DWORD );
        bool CompressBuffer ( InfoBlock&, const LPBYTE );
        bool CollectInfo ( );

        // hardware information
        bool m_bSupportLocking;

        // screen & buffer dimensions
        long m_ScreenHeight;
        long m_ScreenWidth;
        long m_OffscreenHeight;
        long m_OffscreenWidth;
        Rect m_rctScreen;
        Rect m_rctOffscreen;
        long m_padding;
        long m_ByteCount;

        // hardware interface
        Video m_display;

        // buffer size info
        DWORD m_cbTotalBufferSize;
        DWORD m_cbRowBufferSize;
        int m_BitsPerPel;
        HWND m_hWnd;

        // checksum class
        CheckSum m_checksum;

        // compression
        CompressionEngine m_compressionEngine;
};
// adminvideo.h

// August 4, 1997
// Rob Gagne

// manages the admin side of the video transaction

class AdminVideo
{
public:
```

```
    AdminVideo ( );
    ~AdminVideo ( );

    bool OpenSession (  const HardwareInfo&, HWND,
LPPALETTEENTRY );
    void CloseSession ( );
    bool ProcessFrame ( InfoBlock&, DirtyBlock*, LPBYTE,
DWORD );

    long TotalBufferSize ( ) { return m_cbTotalBufferSize;
}
    long GridCount ( )  { return GRID_WIDTH * GRID_HEIGHT;
}
    int  MaxPalSize ( ) { return m_display.MaxPalSize ( );
}
    bool SetPalette ( LPPALETTEENTRY );
    bool RestoreLostSurface ( );

    bool Pause ( )  {   return m_bLost; }
    bool Refresh ( );

private:
    bool ProcessIteration ( InfoBlock&, DirtyBlock*, DWORD
);
    bool ExpandBuffer ( InfoBlock&, LPBYTE );
    void ProcessInfo ( const HardwareInfo& );

    // screen & buffer dimensions
    long m_ScreenHeight;
    long m_ScreenWidth;
    long m_OffscreenHeight;
    long m_OffscreenWidth;
    Rect m_rctScreen;
    Rect m_rctOffscreen;
    long m_padding;
    int m_ByteCount;

    // hardware interface
    Video m_display;
    HWND m_hWnd;

    // buffer size info
    DWORD m_cbTotalBufferSize;
    DWORD m_cbRowBufferSize;
    int m_BitsPerPel;

    // surface lost
```

```
        bool m_bLost;
        bool m_bRefresh;

        // compression
        CompressionEngine m_compressionEngine;
};

inline bool AdminVideo::Refresh ( )
{
        if (m_bRefresh)
        {
                m_bRefresh = false;
                return true;
        }
        return false;
}


// ahuff.h
// header for adaptive huffman class

#define END_OF_STREAM       256
#define ESCAPE              257
#define SYMBOL_COUNT        258
#define NODE_TABLE_COUNT    ( ( SYMBOL_COUNT * 2 ) - 1 )
#define ROOT_NODE           0
#define MAX_WEIGHT          0x8000
#define TRUE                1
#define FALSE               0


class AdaptHuffComp
{
public:
        AdaptHuffComp ( );
        ~AdaptHuffComp ( );

        long CompressBuffer (LPBYTE, LPBYTE, long,
bool=false);
        bool ExpandBuffer (LPBYTE, LPBYTE, long, long,
bool=false);

private:
        // internal structures
        struct Tree
        {
                int leaf[ SYMBOL_COUNT ];
```

```
            int next_free_node;
            struct Node
            {
                  unsigned int weight;
                  int parent;
                  int child_is_leaf;
                  int child;
            } nodes[ NODE_TABLE_COUNT ];
      };
      Tree m_tree;

      void InitializeTree(  );
      void EncodeSymbol( unsigned int c, BIT_MANIP *output
);
      int DecodeSymbol( BIT_MANIP *input );
      void UpdateModel( int c );
      void RebuildTree(  );
      void swap_nodes( int i, int j );
      void add_new_node( int c );

};


//** winsock2 defines **//

#define SD_RECEIVE      0x00
#define SD_SEND         0x01
#define SD_BOTH         0x02


class Except
{
public:
      Except ( LPCTSTR );
      void Trace ( );
private:
      DWORD     m_LastError;
      LPCTSTR m_pError;
};


class BaseSocket
{
public:
      // interface
      BaseSocket ( );
      ~BaseSocket ( );
```

```
      int Send  ( LPBYTE, int ) const;
      int Recv  ( LPBYTE, int ) const;
      int SendFully  ( LPBYTE, int ) const;
      int RecvFully  ( LPBYTE, int ) const;
      void EmptyRecvBuffer ( ) const;
      bool CanRead  ( int = 30 ) const;
      bool CanWrite ( int = 30 ) const;

      void Shutdown ( int=SD_SEND );
      void Close ( );
protected:
      // data
      SOCKET              m_socket;
      sockaddr_in         m_addr;
      int                 m_nPort;
      bool           m_bCreated;
      bool           m_bConnected;

      // protected methods
      void InitClass ( );
      void ResolveName ( int, LPCTSTR );
      void Bind ( int, LPCTSTR=NULL );
      void Create ( );
      bool IPFromAddr ( sockaddr_in*, LPTSTR, int&);
      bool NameFromAddr ( sockaddr_in*, LPTSTR, int&);
      bool IsIpAddr ( LPCTSTR, unsigned char*);

};

class ServerSocket : public BaseSocket
{
public:
      // methods
      ServerSocket ( );
      void Create ( int nPort );
      void Accept ( ServerSocket& );
      void Listen ( );
      bool ClientName ( LPTSTR, int& );
      bool ClientIP ( LPTSTR, int& );
      bool ServerName ( LPTSTR, int& );
      bool ServerIP ( LPTSTR, int& );
private:
      // data
      sockaddr_in         m_client_addr;
      sockaddr_in         m_resolved_name;

      bool ResolveLocalName ( sockaddr_in* );
```

```
};

class ClientSocket : public BaseSocket
{
public:
     void Create ( );
     void Connect ( LPCTSTR, int nPort );
};

// rle.h

#define BYTE_MAX 0xff

long rle_compress ( LPBYTE pIn, LPBYTE pOut, long dwLen );
bool rle_expand ( LPBYTE pIn, LPBYTE pOut, long , long );


// ratio.h

// object for deciding the compression algorithm to use
// based on compression / sending times


#define UPPER_LIMIT      3.00f
#define MID_UPPER_LIMIT  1.50f
#define MID_LOWER_LIMIT 0.67f
#define LOWER_LIMIT      0.30f

#define MAX_NUM 10

const long MAX_COMPRESSION = 4;
const long MIN_COMPRESSION = 0;

class Ratio
{
public:
     Ratio ( );
     ~Ratio ( );

     void SaveCollectionTime       ( DWORD dwT)
          { dwLastCollectionTime = dwT; }
     void SaveSendTime    ( DWORD dwT)
          { dwLastTransmitionTime = dwT; }

     DWORD CompressionScheme ( );
private:
     DWORD dwLastCollectionTime;
```

```
    DWORD dwLastTransmitionTime;
    long  dwCurrentCompression;

    float flAvgRatio;
    int num;

    DWORD arraySchemes [ MAX_COMPRESSION ];
};

// huff.h
// header for non-adaptive huffman compression

// dependencies: bitio.h rle.h

#define END_OF_STREAM 256

class HuffComp
{
public:
    HuffComp ( );
    ~HuffComp ( );

    long CompressBuffer (LPBYTE, LPBYTE, long,
bool=false);
    bool ExpandBuffer (LPBYTE, LPBYTE, long, long,
bool=false);

private:
    // data
    struct CODE
    {
        unsigned int code;
        int code_bits;
    };
    struct NODE
    {
        unsigned int count;
        unsigned int saved_count;
        int child_0;
        int child_1;
    };
    unsigned long *counts;
    NODE *nodes;
    CODE *codes;

    // initializeation
    void CreateTables ( );
```

```
        void CleanupTables ( );
        void InitializeTables ( LPBYTE, long );

        // utility functions
        void count_bytes( LPBYTE, long, unsigned long *);
        void scale_counts( unsigned long *, NODE *);
        int build_tree( NODE *);
        void convert_tree_to_code( NODE *, CODE *, unsigned
int, int, int);
        void output_counts( BIT_MANIP*, NODE *);
        void input_counts(  BIT_MANIP*, NODE *);
        void compress_data( Buffer&, BIT_MANIP *, CODE *);
        void expand_data( BIT_MANIP *, Buffer&, NODE *, int);

};

// hardware.h
// header for the hardware class to contain the direct draw
abstraction

#define MAX_PAL 256

class Video
{
public:
        Video ( );
        ~Video ( );

        // initializing the direct draw system
        // width, height, width, height, client/admin
        bool Open ( long, long, long, long, DWORD fType, int,
                        HWND=NULL, LPPALETTEENTRY=NULL );
        void Close ( );
        bool GetScreenMemory ( RECT*, LPBYTE& );
        bool GetBufferMemory ( RECT*, LPBYTE& );
        // from offscreen to screen
        bool PutScreenRect ( RECT& scrn, RECT& offscrn);
        // from screen to offscreen
        bool GetScreenRect ( RECT& scrn, RECT& offscrn);
        bool RestoreLostSurface ( );
        long GetSurfacePitch ( );
        long GetBufferPitch ( );
        // palette routines
        bool GetEntries ( LPPALETTEENTRY&, int& );
        bool SetEntries ( const LPPALETTEENTRY, int);
        int MaxPalSize  ( ) { return m_PalEntryCount; }
```

```
    bool SupportScreenLocking ( ) {  return
m_bSupportSLock; }

    enum { SCREEN_ADMIN, SCREEN_CLIENT };
private:
    // data interface
    HRESULT m_Result;
    long ScreenWidth;
    long ScreenHeight;
    long OffscreenWidth;
    long OffscreenHeight;
    int BitCount;
    int m_ByteCount;
    HWND m_hWnd;

    // direct draw objects
    LPDIRECTDRAW pDirectDraw;
    LPDIRECTDRAWSURFACE pScreen;
    LPDIRECTDRAWSURFACE pOffscreen;
    LPDIRECTDRAWPALETTE pPalette;

    // palette datastructures
    LPPALETTEENTRY m_pSavedEntries;
    LPPALETTEENTRY m_pCurrentEntries;
    int m_PalEntryCount;

    // private interface
    bool OpenAdmin ( LPPALETTEENTRY=NULL );
    bool OpenClient ( );
    bool OpenPrimarySurface ( );
    bool OpenBackBufferSurface ( );
    bool OpenPalette ( LPPALETTEENTRY = NULL );
    bool InitPaletteBuffers ( );
    bool CompareEntries ( LPPALETTEENTRY );

    // capabilities
    bool m_bSupportSLock;
    bool m_bSupportOLock;


};



class Rect : public RECT
```

```
{
public:
     Rect ( ) {};
     Rect (int Width, int Height, int Rows, int Columns);
     ~Rect ( ) { }
     RECT& MoveNext ( );
     RECT& MovePrev ( );
     RECT& MoveFirst ( );
     RECT& MoveTo (int, int);
     RECT* FullArea ( ) {return &m_FullArea;}
     RECT* CurrentGrid ( ) {return this;}
     int  GridWidth  ( ) {return m_GridW;}
     int GridHeight ( ) {return m_GridH;}
     int PosX ( ) const {return left;}
     int PosY ( ) const {return top; }
     int GridPosX ( ) const {return m_x;}
     int GridPosY ( ) const {return m_y;}
     int GridArea ( ) const {return m_GridArea;}
     bool End ( ) const {return m_bEnd;}
private:
     // static information
     int m_nHeight;
     int m_nWidth;
     int m_nRows;
     int m_nColumns;
     int m_GridW;
     int m_GridH;
     RECT m_FullArea;
     int m_GridArea;
     // dynamic information
     int m_x;
     int m_y;
     bool m_bEnd;

     // helpers
     void SetRect ( );
};


// sets the rect to the current m_x and m_y position
inline void Rect::SetRect ( )
{
     left   = m_x * m_GridW;
     top    = m_y * m_GridH;
     right  = left + m_GridW;
     bottom = top  + m_GridH;
}
```

```
// diag.h
// diagnostic error handling routines

////////////////////////////////////////
// ** Enable or Disable Diagnostics **

#define _DEBUG_OUTPUT_
#define _LOG_TRACE_


// Diagnostics
#ifdef _DEBUG_OUTPUT_
     void Log_Trace (LPCTSTR pMsg);
     void Log_TraceLastError ( );
     bool DebugAssert (int, LPTSTR);
     #define TRACE(pMsg)  Log_Trace(pMsg)
     #define LAST_ERROR() Log_TraceLastError( )
     #define DD_CALL_INIT() HRESULT ddr
     #define DD_CALL(call)    ddr = call; \
                                   if (ddr != DD_OK) TRACE(
DDErrorToString (ddr))
     #define DD_FAIL()    (DD_OK != ddr)
     #define DD_RESULT()  (ddr)
     #define DD_SUCCESS() (DD_OK == ddr)
     #define BOOL_CALL(call) if ((call)==0) LAST_ERROR()
     #define ASSERT(test) ( (test) || DebugAssert(__LINE__,
__FILE__))
     #define TRACE_FAIL() DebugAssert ( __LINE__, __FILE__
)
#else
     #define TRACE(pMsg)
     #define DD_CALL(call)    ddr = (call)
     #define   DD_CALL_INIT() HRESULT ddr
     #define DD_SUCCESS() (DD_OK == ddr)
     #define DD_FAIL()    (DD_OK != ddr)
     #define DD_RESULT()  (ddr)
     #define ASSERT(test)
     #define LAST_ERROR()
     #define BOOL_CALL()
     #define TRACE_FAIL()
#endif

#ifdef _LOG_TRACE_
     extern HANDLE hLogFile;
     void OpenLogFile (LPCTSTR);
```

```
#endif

TCHAR* DDErrorToString(HRESULT error);



// Global Utilities

//void ResourceMessageBox (UINT, UINT=IDS_CAPTION,
DWORD=MB_OK);


// compressionEngine.h
// July 25, 1997
// abstraction of the compression agorithms
// allows compression and expansion using several
algorithms
// hides the implementation details of the algorithms
// provides only buffer to buffer compression

#define CPX_NONE              0x00
#define CPX_CUSTOM_RLE        0x01
#define CPX_HUFFMAN_RLE       0x02
#define CPX_ADAPT_HUFFMAN   0x03
#define CPX_CRUSHER_RLE_9     0x04
#define CPX_CRUSHER_RLE_10    0x05
#define CPX_CRUSHER_RLE_11    0x06
#define CPX_CRUSHER_RLE_12    0x07
#define CPX_CRUSHER_RLE_13    0x08

#define CRUSHER_VERSION 1

class CompressionEngine
{
public:
     CompressionEngine ( );
     ~CompressionEngine ( );

     // interface: In, Out, FullSize, CompressedSize,
Algorithm
     bool Compress ( LPBYTE, LPBYTE, const long, long&,
DWORD );
     bool Expand   ( LPBYTE, LPBYTE, const long, const
long, DWORD);

private:
     HuffComp m_huff;
```

```
    AdaptHuffComp m_Ahuff;

    // Rle wrapping
    LPBYTE m_pRleBuffer;
    long   m_cbCompressed;
    bool m_bFailCrusher;

    bool RleCompressWrapStart ( LPBYTE, LPBYTE&, const
long);
    bool RleCompressWrapFinish ( );
    bool RleExpandWrapStart ( LPBYTE& );
    bool RleExpandWrapFinish ( LPBYTE, long );
};


// TCP/IP ports
#define VIDEO_PORT 4000
#define INPUT_PORT 5000

class Comm
{
public:
    Comm  ( HANDLE );
    ~Comm ( );

    //
    bool Wait ( );
    bool Connect ( LPCTSTR );
    bool PrepareServer ( );
    bool RemoteInfo (LPTSTR, LPTSTR, int);

    // transfer interface
    void VideoSend (LPBYTE, int);
    void VideoRecv (LPBYTE, int);
    void InputSend (LPBYTE, int);
    void InputRecv (LPBYTE, int);
    void Close ( );

    enum { STATUS_LOOKINGUP_NAME, STATUS_AUTHENTICATING };
private:
    BaseSocket *pVSock;
    BaseSocket *pISock;

    ClientSocket   m_ClVideoSocket;
    ClientSocket   m_ClInputSocket;
    ServerSocket   m_SvVideoSocket;
    ServerSocket   m_SvInputSocket;
```

```
     ServerSocket   m_ListenVideo;
     ServerSocket   m_ListenInput;

     HANDLE m_hSignal;
     bool   m_Connected;
};

// checksum.h

#define CX_CRCMASK               0xFFFFFFFFL
#define CX_CRC32_POLYNOMIAL      0xEDB88320L

#define CK_STEP 2;

class CheckSum
{
public:
     CheckSum ( );
     ~CheckSum ( );
     void Initialize (long, long, long, long);
     bool ComputeFullCheckSum (LPDWORD);
     bool ComputeRectCheckSum (LPDWORD, const RECT&, int x,
int y);
     bool Dirty ( int x, int y ) {
          ASSERT ( x < GRID_WIDTH && y < GRID_HEIGHT);
          return (m_dwCurrentCRC [x][y]!=
                    m_dwSavedCRC   [x][y]);
          }
     void Synch ( int x, int y ) {
          ASSERT ( x < GRID_WIDTH && y < GRID_HEIGHT);
          m_dwSavedCRC [x][y] = m_dwCurrentCRC [x][y];
          }
private:

     DWORD m_Length;
     DWORD m_LineLength ;
     DWORD m_MaxLine;
     DWORD m_First;
     DWORD m_Second;
     DWORD m_Pitch;

     DWORD *Ccitt32Table;

     DWORD m_dwCurrentCRC [GRID_WIDTH][GRID_HEIGHT];
     DWORD m_dwSavedCRC   [GRID_WIDTH][GRID_HEIGHT];

     long m_Width;
```

```
     long m_Height;
     long m_ByteCount;

     void InitTable ( );
     void ReleaseTable ( );
     inline DWORD ComputeCheckSum ( DWORD, int );
     DWORD cx_lCRC32Polynomial;
};

inline DWORD CheckSum::ComputeCheckSum (DWORD lCRC, int c)
{
   DWORD  lTemp1 ;
   DWORD  lTemp2 ;

   lTemp1 = ( lCRC >> 8 ) & 0x00FFFFFFL ;
   lTemp2 = Ccitt32Table[ ( (cxINT) lCRC ^ c ) & 0xff ] ;
   lCRC = lTemp1 ^ lTemp2 ;
   return( lCRC ) ;
}


/************************** Start of BITIO.H
*************************/

#ifndef _BITIO_H
#define _BITIO_H


class Buffer
{
public:
     Buffer (LPBYTE p, long cb) : pBeg(p), pCur(p),
cbMaxLen(cb) { pEnd = pBeg + cbMaxLen; }
     void Put (const BYTE);
     void Get (BYTE&);
     void Get (int&);
     void Get (unsigned int&);
     bool End ( ) { return (pCur >= pEnd); }
     void SetToStart ( ) { pCur = pBeg; }
     long CurLen ( ) { return (pCur - pBeg); }
private:
     long cbMaxLen;
     LPBYTE pBeg;
     LPBYTE pCur;
     LPBYTE pEnd;
};
```

```
struct BIT_MANIP
{
     BIT_MANIP (LPBYTE p, long cb) : block (p,cb) { }
     // my fields
     Buffer block;
     // existing fields
     unsigned char mask;
     int rack;
};


BIT_MANIP* OpenOutput ( LPBYTE, DWORD);
BIT_MANIP* OpenInput  ( LPBYTE, DWORD);
long CloseOutput ( BIT_MANIP *);
void CloseInput  ( BIT_MANIP *);
void OutputBit   ( BIT_MANIP * , int);
void OutputBits  ( BIT_MANIP *, unsigned long, int);
int InputBit     ( BIT_MANIP *);
unsigned long InputBits( BIT_MANIP *, int);


long CompressBuffer (LPBYTE pIn, LPBYTE pOut, long cbSize);
void ExpandBuffer ( LPBYTE pIn, LPBYTE pOut, long
cbCompressedSize );


#endif  /* _BITIO_H */

/*************************** End of BITIO.H
*************************/


// systemsettings.h

class SystemSettings
{
public:
     SystemSettings ( );
     ~SystemSettings ( );

     bool Set ( );
     void Restore ( );

     int Height ( ) { return m_nHeight; }
     int Width ( )  { return m_nWidth; }
     int Depth ( )  { return m_CurrentColorDepth; }
private:
```

```
    enum { DESKTOP = 0x01, VIDEO = 0x02 };

    // stored settings to restore
    int                 m_ColorDepth;
    int                 m_CurrentColorDepth;
    DWORD               m_dwDesktopColor;
    DWORD               m_fAltered;
    int                 m_nWidth;
    int                 m_nHeight;

    // private interface
    bool SetDesktop ( );
    bool SetVideo ( );
    void RestoreDesktop ( );
    void RestoreVideo ( );
    void SetCursors ( );
    void RestoreCursors ( );

    DWORD DisplaySuspendStatus;
};


// hardware.cpp
// source file for the DirectDraw hardware abstraction
// July 25, 1997
// by Rob Gagne

#include <windows.h>
#include <tchar.h>
#include "consultant.h"
#include "ddraw.h"
#include "windowed_hardware.h"
#include "diag.h"


WindowedVideo::WindowedVideo ( )
{
    // data interface
    BitCount = 8;

    // direct draw objects
    pDirectDraw = NULL;
    pScreen     = NULL;
    pOffscreen  = NULL;
    pPalette    = NULL;

    m_PalEntryCount = 0;
```

```
    m_pSavedEntries   = NULL;
    m_pCurrentEntries = NULL;
}

WindowedVideo::~WindowedVideo ( )
{
    Close ( );
}

// closing the objects
//////////////////////////////////////

void WindowedVideo::Close ( )
{
    DD_CALL_INIT ( );
    if (pOffscreen)
    {
        DD_CALL (pOffscreen->Release ( ));
        pOffscreen = NULL;
    }
/*
    if (pPalette)
    {
        DD_CALL (pPalette->Release ( ));
        pPalette = NULL;
    }
*/
    if (pScreen)
    {
        DD_CALL (pScreen->Release ( ));
        pScreen = NULL;
    }
    if (pDirectDraw)
    {
        DD_CALL (pDirectDraw->RestoreDisplayMode ( ));
        DD_CALL (pDirectDraw->Release ( ));
        pDirectDraw = NULL;
    }
    if (m_pSavedEntries)
    {
        delete m_pSavedEntries;
        m_pSavedEntries = NULL;
    }
    if (m_pCurrentEntries)
    {
        delete m_pCurrentEntries;
        m_pCurrentEntries = NULL;
```

```
    }
}

bool WindowedVideo::Open ( long w, long h, long off_w, long
off_h, DWORD fMode,
                          HWND hWnd/*=NULL*/, LPPALETTEENTRY
pPal/*=NULL*/)
{
    ScreenWidth = w;
    ScreenHeight = h;
    OffscreenWidth = off_w;
    OffscreenHeight = off_h;
    m_hWnd = hWnd;
    switch (fMode)
    {
    case SCREEN_ADMIN:
        return OpenAdmin ( pPal );
    case SCREEN_CLIENT:
        return OpenClient ( );
    default:
        TRACE ("Bad Mode in Vido::Open\n");
        break;
    }
    return false;
}

////////////////////////////////////////////////////////////
////////////
// creating the direct draw objects

bool WindowedVideo::OpenAdmin ( LPPALETTEENTRY
pPal/*=NULL*/ )
{
    TRACE ( "** Opening Direct Draw Objects as Admin\n" );
    DD_CALL_INIT ( );

    // create direct draw object
    DD_CALL (DirectDrawCreate (NULL, &pDirectDraw, NULL));
    if (DD_FAIL ( ))    return false;

    // set the cooperative level to exclusive
    DD_CALL (pDirectDraw->SetCooperativeLevel (m_hWnd,
DDSCL_NORMAL ));
    if (DD_FAIL ( ))    return false;

    // change the resolution to match the client
    /*
```

```
    DD_CALL (pDirectDraw->SetDisplayMode (
         ScreenWidth, ScreenHeight,    BitCount));
    if (DD_FAIL ( ))    return false;
    */
    if (InitPaletteBuffers ( )    == false) return false;
    if (OpenPrimarySurface ( )    == false) return false;
    if (OpenBackBufferSurface ( ) == false) return false;
    if (OpenPalette ( pPal )   == false) return false;

    TRACE ( "** Direct Draw Objects Open\n" );
    return true;
}

bool WindowedVideo::OpenClient ( )
{
    TRACE ( "** Opening Direct Draw Objects as Admin\n" );
    DD_CALL_INIT ( );
    // create direct draw object
    DD_CALL (DirectDrawCreate (NULL, &pDirectDraw, NULL));
    if (DD_FAIL ( ))    return false;

    // set the cooperative level to normal, we only want
to look at the screen
    DD_CALL (pDirectDraw->SetCooperativeLevel (NULL,
DDSCL_NORMAL));
    if (DD_FAIL ( ))    return false;

    if (InitPaletteBuffers ( )    == false) return false;
    if (OpenPrimarySurface ( )    == false) return false;
    if (OpenBackBufferSurface ( ) == false) return false;

    TRACE ( "** Direct Draw Objects Open\n" );
    return true;
}

bool WindowedVideo::OpenPrimarySurface ( )
{
    DD_CALL_INIT( );
    // create the surface
    DDSURFACEDESC dsc = {0};
    dsc.dwSize = sizeof (dsc);
    dsc.dwFlags = DDSD_CAPS;
    dsc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    DD_CALL (pDirectDraw->CreateSurface (&dsc, &pScreen,
NULL));
    if (DD_FAIL ( ))    return false;
```

```
    // check to see if it supports surface locking
    // current implementation is to fail if it does not
    DDSURFACEDESC SurfaceDesc = {0};
    SurfaceDesc.dwSize = sizeof (SurfaceDesc);
    RECT rect;
    rect.left = rect.top = 0;
    rect.right = ScreenWidth;
    rect.bottom = ScreenHeight;
    TRACE ( "About to lock primary surface\n");
    DD_CALL (pScreen->Lock (&rect, &SurfaceDesc,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
    if (DD_FAIL ( ))
    {
        m_bSupportSLock = false;
        TRACE ("Screen does NOT support locking\n");
    }
    else
    {
        DD_CALL (pScreen->Unlock
(SurfaceDesc.lpSurface));
        m_bSupportSLock = true;
        TRACE ("Screen locking is supported\n");
    }
    return true;
}

bool WindowedVideo::OpenBackBufferSurface ( )
{
    DD_CALL_INIT( );
    // Secondary Buffer for storing the dirty rectangles
    DDSURFACEDESC offdsc = {0};
    offdsc.dwSize = sizeof (offdsc);
    offdsc.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
    offdsc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
DDSCAPS_SYSTEMMEMORY;
    offdsc.dwHeight = OffscreenHeight;
    offdsc.dwWidth  = OffscreenWidth;
    DD_CALL (pDirectDraw->CreateSurface (&offdsc,
&pOffscreen, NULL));
    if (DD_FAIL ( ))    return false;

    // check to see if it supports surface locking
    // current implementation is to fail if it does not
    DDSURFACEDESC SurfaceDesc = {0};
    SurfaceDesc.dwSize = sizeof (SurfaceDesc);
    RECT rect;
    rect.left = rect.top = 0;
```

```
    rect.right  =  OffscreenWidth;
    rect.bottom =  OffscreenHeight;
    DD_CALL (pOffscreen->Lock (&rect, &SurfaceDesc,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
    if (DD_FAIL ( ))
    {
        m_bSupportOLock = false;
        TRACE ("Offscreen Surface does NOT support
locking\n");
    }
    else
    {
        DD_CALL (pOffscreen->Unlock
(SurfaceDesc.lpSurface));
        m_bSupportOLock = true;
        TRACE ("Offscreen locking is supported\n");
    }

    // don't currently support non-locking surfaces
    // if (false == m_bSupportSLock) return false; give it
a try
    if (false == m_bSupportOLock) return false;

    return true;
}

// allocate data for holding the palette ( not the DD
object )
// for the client to keep track of palette changes
// rather then sending a new palette every iteration
bool WindowedVideo::InitPaletteBuffers ( )
{
    m_pSavedEntries   = new PALETTEENTRY[MAX_PAL];
    m_pCurrentEntries = new PALETTEENTRY[MAX_PAL];
    if (m_pSavedEntries && m_pCurrentEntries)
m_PalEntryCount = MAX_PAL;
    return (m_pSavedEntries != NULL && m_pCurrentEntries
!= NULL);
}


// compare palettes, return true if they are the same
bool WindowedVideo::CompareEntries ( LPPALETTEENTRY
pEntries )
{
    for (int n = 0; n < MAX_PAL; n++)
    {
```

```
          if ( (m_pSavedEntries [n].peRed   !=
pEntries[n].peRed  )  ||
               (m_pSavedEntries [n].peBlue  !=
pEntries[n].peBlue )  ||
               (m_pSavedEntries [n].peGreen !=
pEntries[n].peGreen)  ||
               (m_pSavedEntries [n].peFlags !=
pEntries[n].peFlags) )
          {
               return false;
          }
     }
     return true;
}


// gets the direct draw object from the primary surface
// either takes an array of entries or creates one from the
// existing display if none are supplied
bool WindowedVideo::OpenPalette ( LPPALETTEENTRY pEntries
/*=NULL*/)
{
     DD_CALL_INIT ( );
     if (pPalette)
     {
          DD_CALL (pPalette->Release ( ));
          pPalette = NULL;
     }
     if (pScreen)
     {
          TRACE ("Creating Palette\n");
          DD_CALL (pScreen->GetPalette ( &pPalette ));
          if (DD_FAIL ( ) )
          {
               if (NULL == pEntries)
               {
                    HDC hDC = CreateDC ( _T("DISPLAY"),
NULL, NULL, NULL);
                    ZeroMemory ( m_pSavedEntries, sizeof
(PALETTEENTRY) * MAX_PAL);
                    GetSystemPaletteEntries ( hDC, 0,
MAX_PAL, m_pSavedEntries );
                    DeleteDC ( hDC );
                    pEntries = m_pSavedEntries;
               }
```

```
                DD_CALL (pDirectDraw->CreatePalette (
DDPCAPS_8BIT | DDPCAPS_ALLOW256,
                     pEntries, &pPalette, NULL));
                if (pPalette)
                {
                     TRACE ("About to set the palette\n");
                     DD_CALL (pScreen->SetPalette ( pPalette
));
                     if (DD_FAIL ( )) return false;
                }
           }
     }
     return ( pPalette != NULL );
}


// public interface call to get the entries
// fails if there are no changes
bool WindowedVideo::GetEntries ( LPPALETTEENTRY& pEntries,
int& Count )
{
     HDC hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
     if (NULL == hDC) return false;
     UINT nColors = GetSystemPaletteEntries ( hDC, 0,
MAX_PAL, m_pSavedEntries );
     DeleteDC ( hDC );
     pEntries = m_pSavedEntries;
     Count = MAX_PAL;
     return true;
}

// sets the array of palette entries into the current
palette
bool WindowedVideo::SetEntries ( const LPPALETTEENTRY
pEntries, int Count )
{
     DD_CALL_INIT ( );
     ASSERT (pPalette);
     if (pPalette)
     {
          DD_CALL (pPalette->SetEntries ( 0, 0, Count,
pEntries ));
          return DD_SUCCESS ( );
     }
     return false;
}
```

```
////////////////////////////////////////////////////////////
////////////////////////////
// Here lie the manipulation functions


// Blits a rect from the screen to a location in
// the offscreen buffer
bool WindowedVideo::GetScreenRect ( RECT& scrn, RECT&
offscrn )
{
     DD_CALL_INIT ( );
     DD_CALL (pOffscreen->BltFast (
          offscrn.left, offscrn.top,
          pScreen, &scrn,
          DDBLTFAST_WAIT | DDBLTFAST_NOCOLORKEY));
     return (DD_SUCCESS( ));
}

// Blits the rect from the offscreen surface to
// the screen
bool WindowedVideo::PutScreenRect ( RECT& scrn, RECT&
offscrn )
{
     DD_CALL_INIT ( );
     DD_CALL (pScreen->BltFast (
          scrn.left, scrn.top,
          pOffscreen, &offscrn,
          DDBLTFAST_WAIT | DDBLTFAST_NOCOLORKEY));
     return (DD_SUCCESS( ));
}


// surface locking / unlocking
bool WindowedVideo::GetScreenMemory ( RECT* pRect, LPBYTE&
pMem)
{
     ASSERT ( m_bSupportSLock );
     DD_CALL_INIT ( );
     DDSURFACEDESC SurfaceDesc = {0};
     SurfaceDesc.dwSize = sizeof (SurfaceDesc);
     DD_CALL (pScreen->Lock (pRect, &SurfaceDesc,
          DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
     pMem = (LPBYTE)SurfaceDesc.lpSurface;
     DD_CALL (pScreen->Unlock (SurfaceDesc.lpSurface));
     return (pMem != NULL);
}
```

```
bool WindowedVideo::GetBufferMemory ( RECT* pRect, LPBYTE&
pMem )
{
     ASSERT ( m_bSupportOLock );
     DD_CALL_INIT ( );
     DDSURFACEDESC SurfaceDesc = {0};
     SurfaceDesc.dwSize = sizeof (SurfaceDesc);
     DD_CALL (pOffscreen->Lock (pRect, &SurfaceDesc,
          DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
     pMem = (LPBYTE)SurfaceDesc.lpSurface;
     DD_CALL (pOffscreen->Unlock (SurfaceDesc.lpSurface));
     return (pMem != NULL);
}


// restore the surface
bool WindowedVideo::RestoreLostSurface ( )
{
     DD_CALL_INIT ( );
     DD_CALL (pOffscreen->Restore ( ));
     DD_CALL (pScreen->Restore  ( ));
     return (DD_SUCCESS( ));
}

long WindowedVideo::GetSurfacePitch ( )
{
     DD_CALL_INIT ( );
     if ( pScreen )
     {
          DDSURFACEDESC SurfaceDesc = {0};
          SurfaceDesc.dwSize = sizeof (SurfaceDesc);
          DD_CALL ( pScreen->GetSurfaceDesc ( &SurfaceDesc
) );
          return SurfaceDesc.lPitch;
     }
     return 0;
}

long WindowedVideo::GetBufferPitch ( )
{
     DD_CALL_INIT ( );
     if ( pScreen )
     {
          DDSURFACEDESC SurfaceDesc = {0};
          SurfaceDesc.dwSize = sizeof (SurfaceDesc);
          DD_CALL ( pOffscreen->GetSurfaceDesc (
&SurfaceDesc ) );
```

```
            return SurfaceDesc.lPitch;
        }
        return 0;
}

#include <windows.h>
#include <ddraw.h>
#include <tchar.h>
#include "socket.h"
#include "consultant.h"
#include "diag.h"

////////////////////////////////////////////////////////////
///

// translates Direct Draw Error codes

TCHAR* DDErrorToString(HRESULT error)
{
    switch(error)
     {
        case DD_OK:
            /* Also includes D3D_OK and D3DRM_OK */
            return _T("No error.\n\0");
        case DDERR_ALREADYINITIALIZED:
            return _T("This object is already
initialized.\n\0");
        case DDERR_BLTFASTCANTCLIP:
            return _T("Return if a clipper object is
attached to the source surface passed into a BltFast
call.\n\0");
        case DDERR_CANNOTATTACHSURFACE:
            return _T("This surface can not be attached to
the requested surface.\n\0");
        case DDERR_CANNOTDETACHSURFACE:
            return _T("This surface can not be detached
from the requested surface.\n\0");
        case DDERR_CANTCREATEDC:
            return _T("Windows can not create any more
DCs.\n\0");
        case DDERR_CANTDUPLICATE:
            return _T("Can't duplicate primary & 3D
surfaces, or surfaces that are implicitly created.\n\0");
        case DDERR_CLIPPERISUSINGHWND:
            return _T("An attempt was made to set a
cliplist for a clipper object that is already monitoring an
hwnd.\n\0");
```

```
        case DDERR_COLORKEYNOTSET:
            return _T("No src color key specified for this
operation.\n\0");
        case DDERR_CURRENTLYNOTAVAIL:
            return _T("Support is currently not
available.\n\0");
        case DDERR_DIRECTDRAWALREADYCREATED:
            return _T("A DirectDraw object representing
this driver has already been created for this
process.\n\0");
        case DDERR_EXCEPTION:
            return _T("An exception was encountered while
performing the requested operation.\n\0");
        case DDERR_EXCLUSIVEMODEALREADYSET:
            return _T("An attempt was made to set the
cooperative level when it was already set to
exclusive.\n\0");
        case DDERR_GENERIC:
            return _T("Generic failure.\n\0");
        case DDERR_HEIGHTALIGN:
            return _T("Height of rectangle provided is not
a multiple of reqd alignment.\n\0");
        case DDERR_HWNDALREADYSET:
            return _T("The CooperativeLevel HWND has
already been set. It can not be reset while the process has
surfaces or palettes created.\n\0");
        case DDERR_HWNDSUBCLASSED:
            return _T("HWND used by DirectDraw
CooperativeLevel has been subclassed, this prevents
DirectDraw from restoring state.\n\0");
        case DDERR_IMPLICITLYCREATED:
            return _T("This surface can not be restored
because it is an implicitly created surface.\n\0");
        case DDERR_INCOMPATIBLEPRIMARY:
            return _T("Unable to match primary surface
creation request with existing primary surface.\n\0");
        case DDERR_INVALIDCAPS:
            return _T("One or more of the caps bits passed
to the callback are incorrect.\n\0");
        case DDERR_INVALIDCLIPLIST:
            return _T("DirectDraw does not support the
provided cliplist.\n\0");
        case DDERR_INVALIDDIRECTDRAWGUID:
            return _T("The GUID passed to DirectDrawCreate
is not a valid DirectDraw driver identifier.\n\0");
        case DDERR_INVALIDMODE:
```

```
            return _T("DirectDraw does not support the
requested mode.\n\0");
        case DDERR_INVALIDOBJECT:
            return _T("DirectDraw received a pointer that
was an invalid DIRECTDRAW object.\n\0");
        case DDERR_INVALIDPARAMS:
            return _T("One or more of the parameters passed
to the function are incorrect.\n\0");
        case DDERR_INVALIDPIXELFORMAT:
            return _T("The pixel format was invalid as
specified.\n\0");
        case DDERR_INVALIDPOSITION:
            return _T("Returned when the position of the
overlay on the destination is no longer legal for that
destination.\n\0");
        case DDERR_INVALIDRECT:
            return _T("Rectangle provided was
invalid.\n\0");
        case DDERR_LOCKEDSURFACES:
            return _T("Operation could not be carried out
because one or more surfaces are locked.\n\0");
        case DDERR_NO3D:
            return _T("There is no 3D present.\n\0");
        case DDERR_NOALPHAHW:
            return _T("Operation could not be carried out
because there is no alpha accleration hardware present or
available.\n\0");
        case DDERR_NOBLTHW:
            return _T("No blitter hardware present.\n\0");
        case DDERR_NOCLIPLIST:
            return _T("No cliplist available.\n\0");
        case DDERR_NOCLIPPERATTACHED:
            return _T("No clipper object attached to
surface object.\n\0");
        case DDERR_NOCOLORCONVHW:
            return _T("Operation could not be carried out
because there is no color conversion hardware present or
available.\n\0");
        case DDERR_NOCOLORKEY:
            return _T("Surface doesn't currently have a
color key\n\0");
        case DDERR_NOCOLORKEYHW:
            return _T("Operation could not be carried out
because there is no hardware support of the destination
color key.\n\0");
        case DDERR_NOCOOPERATIVELEVELSET:
```

```
            return _T("Create function called without
DirectDraw object method SetCooperativeLevel being
called.\n\0");
        case DDERR_NODC:
            return _T("No DC was ever created for this
surface.\n\0");
        case DDERR_NODDROPSHW:
            return _T("No DirectDraw ROP hardware.\n\0");
        case DDERR_NODIRECTDRAWHW:
            return _T("A hardware-only DirectDraw object
creation was attempted but the driver did not support any
hardware.\n\0");
        case DDERR_NOEMULATION:
            return _T("Software emulation not
available.\n\0");
        case DDERR_NOEXCLUSIVEMODE:
            return _T("Operation requires the application
to have exclusive mode but the application does not have
exclusive mode.\n\0");
        case DDERR_NOFLIPHW:
            return _T("Flipping visible surfaces is not
supported.\n\0");
        case DDERR_NOGDI:
            return _T("There is no GDI present.\n\0");
        case DDERR_NOHWND:
            return _T("Clipper notification requires an
HWND or no HWND has previously been set as the
CooperativeLevel HWND.\n\0");
        case DDERR_NOMIRRORHW:
            return _T("Operation could not be carried out
because there is no hardware present or available.\n\0");
        case DDERR_NOOVERLAYDEST:
            return _T("Returned when GetOverlayPosition is
called on an overlay that UpdateOverlay has never been
called on to establish a destination.\n\0");
        case DDERR_NOOVERLAYHW:
            return _T("Operation could not be carried out
because there is no overlay hardware present or
available.\n\0");
        case DDERR_NOPALETTEATTACHED:
            return _T("No palette object attached to this
surface.\n\0");
        case DDERR_NOPALETTEHW:
            return _T("No hardware support for 16 or 256
color palettes.\n\0");
        case DDERR_NORASTEROPHW:
```

```
            return _T("Operation could not be carried out
because there is no appropriate raster op hardware present
or available.\n\0");
        case DDERR_NOROTATIONHW:
            return _T("Operation could not be carried out
because there is no rotation hardware present or
available.\n\0");
        case DDERR_NOSTRETCHHW:
            return _T("Operation could not be carried out
because there is no hardware support for stretching.\n\0");
        case DDERR_NOT4BITCOLOR:
            return _T("DirectDrawSurface is not in 4 bit
color palette and the requested operation requires 4 bit
color palette.\n\0");
        case DDERR_NOT4BITCOLORINDEX:
            return _T("DirectDrawSurface is not in 4 bit
color index palette and the requested operation requires 4
bit color index palette.\n\0");
        case DDERR_NOT8BITCOLOR:
            return _T("DirectDrawSurface is not in 8 bit
color mode and the requested operation requires 8 bit
color.\n\0");
        case DDERR_NOTAOVERLAYSURFACE:
            return _T("Returned when an overlay member is
called for a non-overlay surface.\n\0");
        case DDERR_NOTEXTUREHW:
            return _T("Operation could not be carried out
because there is no texture mapping hardware present or
available.\n\0");
        case DDERR_NOTFLIPPABLE:
            return _T("An attempt has been made to flip a
surface that is not flippable.\n\0");
        case DDERR_NOTFOUND:
            return _T("Requested item was not found.\n\0");
        case DDERR_NOTLOCKED:
            return _T("Surface was not locked.  An attempt
to unlock a surface that was not locked at all, or by this
process, has been attempted.\n\0");
        case DDERR_NOTPALETTIZED:
            return _T("The surface being used is not a
palette-based surface.\n\0");
        case DDERR_NOVSYNCHW:
            return _T("Operation could not be carried out
because there is no hardware support for vertical blank
synchronized operations.\n\0");
        case DDERR_NOZBUFFERHW:
```

```
            return _T("Operation could not be carried out
because there is no hardware support for zbuffer
blitting.\n\0");
        case DDERR_NOZOVERLAYHW:
            return _T("Overlay surfaces could not be z
layered based on their BltOrder because the hardware does
not support z layering of overlays.\n\0");
        case DDERR_OUTOFCAPS:
            return _T("The hardware needed for the
requested operation has already been allocated.\n\0");
        case DDERR_OUTOFMEMORY:
            return _T("DirectDraw does not have enough
memory to perform the operation.\n\0");
        case DDERR_OUTOFVIDEOMEMORY:
            return _T("DirectDraw does not have enough
memory to perform the operation.\n\0");
        case DDERR_OVERLAYCANTCLIP:
            return _T("The hardware does not support
clipped overlays.\n\0");
        case DDERR_OVERLAYCOLORKEYONLYONEACTIVE:
            return _T("Can only have ony color key active
at one time for overlays.\n\0");
        case DDERR_OVERLAYNOTVISIBLE:
            return _T("Returned when GetOverlayPosition is
called on a hidden overlay.\n\0");
        case DDERR_PALETTEBUSY:
            return _T("Access to this palette is being
refused because the palette is already locked by another
thread.\n\0");
        case DDERR_PRIMARYSURFACEALREADYEXISTS:
            return _T("This process already has created a
primary surface.\n\0");
        case DDERR_REGIONTOOSMALL:
            return _T("Region passed to
Clipper::GetClipList is too small.\n\0");
        case DDERR_SURFACEALREADYATTACHED:
            return _T("This surface is already attached to
the surface it is being attached to.\n\0");
        case DDERR_SURFACEALREADYDEPENDENT:
            return _T("This surface is already a dependency
of the surface it is being made a dependency of.\n\0");
        case DDERR_SURFACEBUSY:
            return _T("Access to this surface is being
refused because the surface is already locked by another
thread.\n\0");
        case DDERR_SURFACEISOBSCURED:
```

```
            return _T("Access to surface refused because
the surface is obscured.\n\0");
        case DDERR_SURFACELOST:
            return _T("Access to this surface is being
refused because the surface memory is gone. The
DirectDrawSurface object representing this surface should
have Restore called on it.\n\0");
        case DDERR_SURFACENOTATTACHED:
            return _T("The requested surface is not
attached.\n\0");
        case DDERR_TOOBIGHEIGHT:
            return _T("Height requested by DirectDraw is
too large.\n\0");
        case DDERR_TOOBIGSIZE:
            return _T("Size requested by DirectDraw is too
large, but the individual height and width are OK.\n\0");
        case DDERR_TOOBIGWIDTH:
            return _T("Width requested by DirectDraw is too
large.\n\0");
        case DDERR_UNSUPPORTED:
            return _T("Action not supported.\n\0");
        case DDERR_UNSUPPORTEDFORMAT:
            return _T("FOURCC format requested is
unsupported by DirectDraw.\n\0");
        case DDERR_UNSUPPORTEDMASK:
            return _T("Bitmask in the pixel format
requested is unsupported by DirectDraw.\n\0");
        case DDERR_VERTICALBLANKINPROGRESS:
            return _T("Vertical blank is in
progress.\n\0");
        case DDERR_WASSTILLDRAWING:
            return _T("Informs DirectDraw that the previous
Blt which is transfering information to or from this
Surface is incomplete.\n\0");
        case DDERR_WRONGMODE:
            return _T("This surface can not be restored
because it was created in a different mode.\n\0");
        case DDERR_XALIGN:
            return _T("Rectangle provided was not
horizontally aligned on required boundary.\n\0");
        default:
                //{
                //   TCHAR strError [20];
                //   wsprintf ( strError, "direct draw error
= %lu\n", error & 0xffff);
                //   TRACE (strError);
                //}
```

```
            return _T("Unrecognized error value.\n\0");
    }
}

////////////////////////////////////////////////////////////
////
// bitmap header uitlities, for now

void CreateBitmapHeader ( BITMAPINFO** bmh, RECT*
pRect/*=NULL*/ )
{
     HDC hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
     UINT uNumColors = GetDeviceCaps (hDC, SIZEPALETTE);
     UINT sizePal = 0;
     if (uNumColors > 0)
     {
          sizePal = (uNumColors * sizeof (RGBQUAD)) -
sizeof (RGBQUAD);
     }
     *bmh = (BITMAPINFO*)new BYTE [sizeof (BITMAPINFO) +
sizePal];

     ZeroMemory (&(*bmh)->bmiHeader, sizeof
(BITMAPINFOHEADER));
     (*bmh)->bmiHeader.biSize      = sizeof
(BITMAPINFOHEADER);
     (*bmh)->bmiHeader.biWidth          = GetDeviceCaps
(hDC, HORZRES);
     (*bmh)->bmiHeader.biHeight         = GetDeviceCaps
(hDC, VERTRES);
     (*bmh)->bmiHeader.biPlanes         = GetDeviceCaps
(hDC, PLANES);
     (*bmh)->bmiHeader.biBitCount  = GetDeviceCaps (hDC,
BITSPIXEL);
     (*bmh)->bmiHeader.biCompression    = BI_RGB;
     (*bmh)->bmiHeader.biClrUsed   = (uNumColors > 0) ?
uNumColors : 0;

     if (uNumColors > 0)
     {
          PALETTEENTRY * pEntries = (PALETTEENTRY*)new BYTE
               [sizeof (PALETTEENTRY) * uNumColors];
          GetSystemPaletteEntries ( hDC, 0, uNumColors,
pEntries);
          for (int n = 0; n < (int)uNumColors; n++)
          {
```

```
                (*bmh)->bmiColors[n].rgbRed        =
pEntries[n].peRed;
                (*bmh)->bmiColors[n].rgbBlue  =
pEntries[n].peBlue;
                (*bmh)->bmiColors[n].rgbGreen =
pEntries[n].peGreen;
                (*bmh)->bmiColors[n].rgbReserved = 0;
           }

     }
     DeleteDC (hDC);

     if (pRect)
     {
          pRect->right    = (*bmh)->bmiHeader.biWidth;
          pRect->bottom  = (*bmh)->bmiHeader.biHeight;
          pRect->left = pRect->top = 0;
     }
}

void CreateBitmapFileHeader ( const BITMAPINFO& bmh,
BITMAPFILEHEADER* bmf )
{
     ZeroMemory (bmf, sizeof (BITMAPFILEHEADER));
     bmf->bfType = DIB_HEADER_MARKER;  // "BM";
     bmf->bfSize = sizeof (bmh) + (bmh.bmiHeader.biWidth
          * ((bmh.bmiHeader.biHeight)) *
(bmh.bmiHeader.biBitCount/8)) ;
     bmf->bfOffBits = sizeof (BITMAPFILEHEADER) +
          sizeof (bmh) + (bmh.bmiHeader.biClrUsed * sizeof
(RGBQUAD));
}


void ResourceMessageBox (UINT uMsg, UINT
uCaption/*=IDS_CAPTION*/, DWORD dwStyle/*=MB_OK*/)
{
     static HANDLE hMod = GetModuleHandle ( NULL );
     TCHAR strMsg    [STATIC_BUFFER];
     TCHAR strCaption [STATIC_BUFFER];
     LoadString ( hMod, uMsg,  strMsg,   STATIC_BUFFER);
     LoadString ( hMod, uCaption, strCaption,
STATIC_BUFFER);
     MessageBox ( NULL, strMsg, strCaption, dwStyle);
}
```

```
#include <windows.h>
#include <tchar.h>
#include "systemsettings.h"
#include "diag.h"
#include "consultant.h"


// array containing the registry name,
// resource name and default id
struct Cursor
{
     LPCTSTR pName;
     LPCTSTR SysId;
     DWORD dwCursorId;
} g_Cursors [] = {
     { _T("Arrow"),       IDC_ARROW,          OCR_NORMAL },
     { _T("IBeam"),       IDC_IBEAM,          OCR_IBEAM  },
     { _T("Wait"),        IDC_WAIT,      OCR_WAIT },
     { _T("Crosshair"),   IDC_CROSS,          OCR_CROSS },
     { _T("SizeAll"),     IDC_SIZE,      OCR_SIZEALL },
     { _T("SizeNESW"),    IDC_SIZENESW,  OCR_SIZENESW },
     { _T("SizeNWSE"),    IDC_SIZENWSE,  OCR_SIZENWSE },
     { _T("SizeWE"),           IDC_SIZEWE,
     OCR_SIZEWE },
     { _T("SizeNS"),           IDC_SIZENS,
     OCR_SIZENS },
     { _T("No"),               IDC_NO,             OCR_NO },
     { _T("AppStarting"), IDC_APPSTARTING, OCR_APPSTARTING
}
};


SystemSettings::SystemSettings ( )
{
     m_fAltered = 0;
}

SystemSettings::~SystemSettings ( )
{
     Restore ( );
}

bool SystemSettings::Set ( )
{
     if ( SetDesktop ( ) == false ) return false;
     if ( SetVideo   ( ) == false ) return false;
     return true;
```

```
}

void SystemSettings::Restore ( )
{
     if ( m_fAltered & DESKTOP ) RestoreDesktop ( );
     if ( m_fAltered & VIDEO   ) RestoreVideo ( );
}

bool SystemSettings::SetDesktop ( )
{
     // currently two components that can be changed
     // with independent possibilities of success
     SystemParametersInfo ( SPI_SETDESKWALLPAPER, 0,
"none", SPIF_SENDCHANGE );
     SystemParametersInfo ( SPI_SETDESKPATTERN  , 0,
"none", SPIF_SENDCHANGE );
     m_dwDesktopColor = GetSysColor (COLOR_DESKTOP);
     INT fElement = COLOR_DESKTOP;
     COLORREF cColor = RGB (0, 0, 0);
     SetSysColors (1, &fElement, &cColor);



     // this code attempts to disable the poweroff EPA
stuff
     // different OS / hardware combinatinos seem to do it
     // differently
     /*
     DWORD dwSize = 0;
     HKEY hKey;
     RegOpenKey ( HKEY_CURRENT_USER, _T("Control
Panel\\Desktop"), &hKey);
     if (RegQueryValueEx ( hKey,
_T("ScreenSavePowerOffActive"), NULL, NULL,
          NULL, &dwSize) == ERROR_SUCCESS)
     {
          RegQueryValueEx ( hKey,
_T("ScreenSavePowerOffActive"), NULL, NULL, (LPBYTE)
               &DisplaySuspendStatus,   &dwSize);
          DWORD dwZero = 0;
          RegSetValueEx ( hKey,
_T("ScreenSavePowerOffActive"), NULL, REG_DWORD, (LPBYTE)
&dwZero,
               sizeof(DWORD));
     }
     RegCloseKey (hKey);
     */
```

```
    SetCursors ( );
    PostMessage (HWND_BROADCAST, WM_SETTINGCHANGE, 0,
NULL);
    // add the user succes bit
    m_fAltered |= DESKTOP;
    return true;
}

bool SystemSettings::SetVideo ( )
{
    // change the display to 8 bpp
    DEVMODE dv = {0};
    dv.dmSize           = sizeof (dv);
    dv.dmBitsPerPel          = 8;
    dv.dmFields             = DM_BITSPERPEL;
    HDC hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
    if (NULL == hDC) return false;

    m_nWidth     = GetDeviceCaps (hDC, HORZRES);
    m_nHeight  = GetDeviceCaps (hDC, VERTRES);
    m_ColorDepth = GetDeviceCaps (hDC, BITSPIXEL);

    DeleteDC (hDC);

    // try to change to 8bpp
    bool bChanged = false;
    __try {
        if (ChangeDisplaySettings (&dv, 0) ==
DISP_CHANGE_SUCCESSFUL)
            bChanged = true;
    }
    __except ( true ) {
        TRACE ("Call to video driver crashed\n");
    }

    // currently only support 8 bpp
    /*
    if (false == bChanged ) // && 8 != m_ColorDepth)
    {
        TRACE ("Something bad happened.\n");
        return false;
    }
    */
    hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
    if (NULL == hDC) return false;
```

```
    m_nWidth     = GetDeviceCaps (hDC, HORZRES);
    m_nHeight  = GetDeviceCaps (hDC, VERTRES);
    m_CurrentColorDepth = GetDeviceCaps (hDC, BITSPIXEL);

    DeleteDC (hDC);


    // add the system bit so we know what to restore
    m_fAltered |= VIDEO;
    return true;
}

void SystemSettings::RestoreDesktop ( )
{
    // restore the display, but only if we successfully
changed
    ASSERT ( m_fAltered & DESKTOP );
    m_fAltered ^= DESKTOP;
    LPTSTR strWallpaper;
    DWORD dwSize = 0;
    HKEY hKey;
    RegOpenKey ( HKEY_CURRENT_USER, _T("Control
Panel\\Desktop"), &hKey);
    if (RegQueryValueEx ( hKey, _T("Wallpaper"), NULL,
NULL, NULL, &dwSize) == ERROR_SUCCESS)
    {
        strWallpaper = new TCHAR[dwSize];
        RegQueryValueEx ( hKey, _T("Wallpaper"), NULL,
NULL, (LPBYTE)strWallpaper, &dwSize);
        SystemParametersInfo ( SPI_SETDESKWALLPAPER, 0,
strWallpaper, SPIF_SENDCHANGE );
        delete strWallpaper;
    }

    dwSize = 0;
    if (RegQueryValueEx ( hKey, _T("Pattern"), NULL, NULL,
NULL, &dwSize) == ERROR_SUCCESS)
    {
        strWallpaper = new TCHAR[dwSize];
        RegQueryValueEx ( hKey, _T("Pattern"), NULL,
NULL, (LPBYTE)strWallpaper, &dwSize);
        SystemParametersInfo ( SPI_SETDESKPATTERN  , 0,
strWallpaper, SPIF_SENDCHANGE );
        delete strWallpaper;
    }

    /*
```

```
     if (RegQueryValueEx ( hKey,
_T("ScreenSavePowerOffActive"), NULL, NULL,
          NULL, &dwSize) == ERROR_SUCCESS)
     {
          RegSetValueEx ( hKey,
_T("ScreenSavePowerOffActive"), NULL, REG_DWORD,
               (LPBYTE)&DisplaySuspendStatus,
sizeof(DWORD));
     }
     */
     RegCloseKey (hKey);

     RestoreCursors ( );
     INT fElement = COLOR_DESKTOP;
     SetSysColors (1, &fElement,
(COLORREF*)&m_dwDesktopColor);
     PostMessage (HWND_BROADCAST, WM_SETTINGCHANGE, 0,
NULL);
}

void SystemSettings::RestoreVideo ( )
{
     // restore the resolution to whatever it was before
     ASSERT ( m_fAltered & VIDEO );
     m_fAltered ^= VIDEO;
     DEVMODE dv = {0};
     dv.dmSize           = sizeof (dv);
     dv.dmBitsPerPel          = m_ColorDepth;
     dv.dmFields             = DM_BITSPERPEL;
     ChangeDisplaySettings (&dv, 0);
}


void SystemSettings::SetCursors ( )
{
     // turn off all fancy mouse cursors

     int nNum = sizeof(g_Cursors) / sizeof(Cursor);
     HCURSOR hCursor;

     for (int n = 0; n < nNum; n++)
     {
          hCursor = LoadCursor (NULL, g_Cursors[n].SysId);
          SetSystemCursor (hCursor,
g_Cursors[n].dwCursorId);
     }
}
```

```
void SystemSettings::RestoreCursors ( )
{
     // reset the cursors back to what they were before
     // do this by checking with teh registry settings

     int nNum = sizeof(g_Cursors) / sizeof(Cursor);
     HKEY hKey;
     HCURSOR hCursor;
     DWORD dwSize;

     RegOpenKey ( HKEY_CURRENT_USER, _T("Control
Panel\\Cursors"), &hKey);
     for (int n = 0; n < nNum; n++)
     {
          if (RegQueryValueEx ( hKey, g_Cursors[n].pName,
NULL, NULL, NULL, &dwSize) == ERROR_SUCCESS
               && (dwSize > 0) )
          {
               LPTSTR strCursor = new TCHAR[dwSize];
               RegQueryValueEx ( hKey, g_Cursors[n].pName,
NULL, NULL, (LPBYTE)strCursor, &dwSize);

               hCursor = LoadImage ( NULL, strCursor,
IMAGE_CURSOR, 0, 0,
                    LR_DEFAULTSIZE | LR_LOADFROMFILE );

               SetSystemCursor (hCursor,
g_Cursors[n].dwCursorId);

               delete strCursor;
          }
     }
}
```

```
// AgentClass.cpp
#define INITGUID

#include <windows.h>
#include <ddraw.h>
#include <tchar.h>
```

```
#include "crusher.h"
#include "consultant.h"
#include "resource.h"
#include "socket.h"
#include "rle.h"
#include "diag.h"
#include "bitio.h"
#include "huff.h"
#include "ahuff.h"
#include "compress.h"
#include "ratio.h"
#include "agent.h"
#include "gRect.h"
#include "hardware.h"
#include "checksum.h"
#include "clientvideo.h"
#include "systemsettings.h"
#include "comm.h"

AgentConnection::AgentConnection ( )
{
     m_offset = 0;
     m_hIconOn = LoadIcon (GetModuleHandle (NULL),
MAKEINTRESOURCE(IDI_ON)   );
     m_hIconOff     = LoadIcon (GetModuleHandle (NULL),
MAKEINTRESOURCE(IDI_OFF)  );
     m_hIconWait    = LoadIcon (GetModuleHandle (NULL),
MAKEINTRESOURCE(IDI_WAIT) );
     m_hSignal = CreateEvent (NULL, true, false,
AGENT_EVENT);
     m_hAccept = CreateEvent (NULL, false, false,
AGENT_ALLOW_EVENT);
     //SetCursors ( );

     m_bLogResults = false;
     m_fCompressionAlgorithm = CPX_HUFFMAN_RLE;
     m_fStatus = CONNECTION_TRANSFER;

     m_hSendStart  = CreateEvent ( NULL, false, false,
AGENT_SEND_START );
     m_hSendFinish = CreateEvent ( NULL, false, false,
AGENT_SEND_FINISH );

     m_bPaletteChanged = true;
}
```

```
AgentConnection::~AgentConnection ( )
{
     DestroyIcon (m_hIconOn);
     DestroyIcon (m_hIconOff);
     DestroyIcon (m_hIconWait);
     CloseHandle (m_hSignal);

}

void AgentConnection::CreateControlDialog ( )
{
     m_hDlg = CreateDialog  (GetModuleHandle (NULL),
MAKEINTRESOURCE (IDD_CLIENT),
                                        NULL, (FARPROC)
AgentDlgProc);
     PostMessage (GetDlgItem (m_hDlg, IDC_2), BM_SETCHECK,
(WPARAM)true, 0);
     SetWindowText (GetDlgItem (m_hDlg, IDC_BUILD),
__TIMESTAMP__);
}


bool AgentConnection::InitVideoLoop ( )
{
     if (VideoWait ( ) == false)
     {
          Disconnect ( );
          return false;
     }
     TRACE ("Video Wait successful.\n");
     if (VideoHandshake ( ) == false)
     {
        Disconnect ( );
          return false;
     }
     TRACE ("Video Handshake successful.\n");
     return true;
}


//bool AgentConnection::


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

```
int AgentConnection::GridVideoLoop ( )
{
     TRACE ("Entering (grid) VideoLoop\n");
     Sleep ( 2000 );
     // locals
     HardwareInfo   info;
     InfoBlock Header [2];
     bool bInit = false;
     bool m_bPal;

     if (InitVideoLoop ( )          == false) return
VIDEO_EXIT_HANDSHAKE_ERROR;

//   if (bInit = system.Set ( )  == false)
//   {
//        info.SetFail ( );
//        VideoSend ( (LPBYTE)&info, sizeof (info));
//        Disconnect ( );
//        return VIDEO_EXIT_HARDWARE_ERROR;
//   }

     ClientVideo video;
     if (video.OpenSession ( m_hDlg ) == false)
     {
          info.SetFail ( );
          VideoSend ( (LPBYTE)&info, sizeof (info));
          Disconnect ( );
          return VIDEO_EXIT_HARDWARE_ERROR;
     }

     // make two buffers for dual threading
     DirtyBlock *pDirtyArray [2];
     LPBYTE pCompressedArray [2];
     LPPALETTEENTRY pPal     [2];
     pDirtyArray [0] =     new DirtyBlock [video.GridCount
( ) ];
     pDirtyArray [1] =     new DirtyBlock [video.GridCount
( ) ];
     pCompressedArray[0] = new BYTE [video.TotalBufferSize
( )];
     pCompressedArray[1] = new BYTE [video.TotalBufferSize
( )];
     InitializeSendThread ( );

     // send over prelim information
     video.QueryHardware (info);
     VideoSend ( (LPBYTE)&info, sizeof (info));
```

```
     m_bPal = ( info.ByteCount == 1 );
     if ( m_bPal ) TRACE ( "Palettized\n" );
     else TRACE  ( "Not-Palettized\n" );

     if ( m_bPal )
     {
          pPal [0]            = new PALETTEENTRY
[video.MaxPalSize ( )];
          pPal [1]            = new PALETTEENTRY
[video.MaxPalSize ( )];
          video.GetPalette    ( Header[0], pPal [0] );
          VideoSend ( (LPBYTE)pPal[0],   256 * sizeof
(PALETTEENTRY));
     }

     // ** compression statistic variables ** //
     DWORD dwStart, dwEnd, dwCollectionEnd;
     TCHAR strResult [255];

     DWORD fCommands = ClientVideo::FORCE_PAINT;
     bool bContinue = true;
     bool Cur = 0;
     int nIterations = 0;
     Status status;

     while (bContinue)
     {
          dwStart = GetTickCount ( );

          if (WaitForSingleObject (m_hSignal, 0) ==
WAIT_OBJECT_0) bContinue = false;

          if (false == bContinue)
          {
               Header[Cur].fStatus = (bContinue ?
VIDEO_NO_PAINT : VIDEO_CLOSE_CONNECTION);
               VideoSend ((LPBYTE)&Header[Cur],
sizeof(Header[Cur]));
               continue;
          }

          // testing purposes, allow manual setting
          // Header[Cur].fCompression =
m_fCompressionAlgorithm;
```

```
          Header[Cur].fCompression =
m_scheme.CompressionScheme ( );

          video.ProcessFrame ( Header[Cur],
pDirtyArray[Cur], pCompressedArray[Cur], fCommands );
          if ( m_bPal && m_bPaletteChanged )
          {
               video.GetPalette    ( Header[Cur], pPal [Cur]
);
               if (nIterations > 10) m_bPaletteChanged =
false;
          }

          dwCollectionEnd    = GetTickCount ( );

          // update the ui (temporary) to show what
compression is used

          if (m_fCompressionAlgorithm !=
Header[Cur].fCompression)
          {
               m_fCompressionAlgorithm =
Header[Cur].fCompression;
               UpdateCompressionUI ( );
          }


          // transfer data to thread for sending
          WaitForSingleObject ( m_hReadyToSend, WAIT_TIME
);
          fCommands = 0;
          VideoRecv ( (LPBYTE)&status, sizeof (status) );
          if (status.Refresh ( ))
          {
               TRACE ("Received a refresh signal\n");
               fCommands |= ClientVideo::FORCE_PAINT;
          }


          m_scheme.SaveCollectionTime ( dwCollectionEnd -
dwStart );

          m_pTxDirtyArray               = pDirtyArray
[Cur];
          m_pTxCompressedBlock     = pCompressedArray
[Cur];
          m_pTxheader                   = &Header [Cur];
```

```
            m_pTxPal                    = pPal[Cur];
            Cur = (!Cur);
            SetEvent ( m_hDataReady );
            // send the thread on it's way

            dwEnd   = GetTickCount ( );
            if (m_bLogResults)
            {
                 wsprintf ( strResult, "Total iteration: %lu
, collection %lu\n", (dwEnd-dwStart),
                      (dwCollectionEnd - dwStart) );
                 TRACE (strResult);
            }

            nIterations ++;

      }

      WaitForSingleObject ( m_hSendThread, WAIT_TIME );

      delete[] pDirtyArray [0];
      delete[] pDirtyArray [1];
      delete[] pCompressedArray[0];
      delete[] pCompressedArray[1];
      if ( m_bPal )
      {
           delete[] pPal [0];
           delete[] pPal [1];
      }
      TRACE ("Exiting Video Thread.\n");
      return VIDEO_EXIT_SUCCESS;
}

/*
int AgentConnection::GridVideoLoop ( )
{
      TRACE ("Entering (grid) VideoLoop\n");

      // locals
      HardwareInfo   info;
      SystemSettings system;
      InfoBlock Header [2];
      bool bInit = false;

      if (InitVideoLoop ( )          == false) return
VIDEO_EXIT_HANDSHAKE_ERROR;
//    if (bInit = system.Set ( )   == false)
```

```
//    {
//         info.SetFail ( );
//         VideoSend ( (LPBYTE)&info, sizeof (info));
//         Disconnect ( );
//         return VIDEO_EXIT_HARDWARE_ERROR;
//    }
      ClientVideo video;
      if (video.OpenSession ( m_hDlg ) == false)
      {
           info.SetFail ( );
           VideoSend ( (LPBYTE)&info, sizeof (info));
           Disconnect ( );
           return VIDEO_EXIT_HARDWARE_ERROR;
      }

      // send over prelim information
      video.QueryHardware (info);
      VideoSend ( (LPBYTE)&info, sizeof (info));

      // allocate buffers
      DirtyBlock* arrayDirty =       new DirtyBlock
[info.MaxGridCount];
      LPBYTE pCompressedBlock = new BYTE
[video.TotalBufferSize ( )];

      //InitializeSendThread ( );

      // ** compression statistic variables ** //
      DWORD dwStart, dwEnd, dwCompressionStart,
dwCompressionEnd;
      TCHAR strResult [255];

      DWORD fCommands = ClientVideo::FORCE_PAINT;
      bool bContinue = true;
      bool Cur = 0;

      while (bContinue)
      {
           dwStart = GetTickCount ( );

           if (WaitForSingleObject (m_hSignal, 0) ==
WAIT_OBJECT_0) bContinue = false;

           if (false == bContinue || CONNECTION_PAUSE ==
m_fStatus )
           {
```

```
                Header[Cur].fStatus = (bContinue ?
VIDEO_NO_PAINT : VIDEO_CLOSE_CONNECTION);
                VideoSend ((LPBYTE)&Header[Cur],
sizeof(Header[Cur]));
                fCommands |= ClientVideo::FORCE_PAINT;
          }



          Header[Cur].fCompression =
m_fCompressionAlgorithm;
          if ( video.ProcessFrame ( Header[Cur],
arrayDirty, pCompressedBlock, fCommands ) == false)
          {
                VideoSend ((LPBYTE)&Header[Cur],
sizeof(InfoBlock));
                continue;
          }
          // all systems go, send the stuff
          VideoSend ( (LPBYTE)&Header[Cur],
sizeof(Header[Cur]));
          VideoSend ( (LPBYTE)arrayDirty,
Header[Cur].nDirtyCount * sizeof (DirtyBlock));
          VideoSend ( pCompressedBlock,
Header[Cur].cbCompressedSize);

          //dwSendEnd = GetTickCount ( );
          if (m_bLogResults)
          {
                //wsprintf (strResult,
                //   "Cx Time: %lu Send Time: %lu  Dirty:
%lu FullSize: %lu CompSize: %lu\n",
                //   (dwEnd-dwStart), (dwSendEnd - dwEnd),
                //   header.nDirtyCount, header.cbFullSize,
header.cbCompressedSize);
                wsprintf ( strResult, "Total iteration: %lu
, compression %lu\n", (dwEnd-dwStart),
                      (dwCompressionEnd - dwCompressionStart)
);
                TRACE (strResult);
          }

          fCommands ^= ClientVideo::FORCE_PAINT;

     }
     // delete pCompressedBlock;
     delete arrayDirty;
```

```
        delete pCompressedBlock;

        TRACE ("Exiting Video Thread.\n");
        return VIDEO_EXIT_SUCCESS;
}
*/


// send thread

void AgentConnection::SendProxy ( AgentConnection* pThis )
{
        pThis->SendThread ( );
}

void AgentConnection::InitializeSendThread ( )
{
        DWORD dwThreadID;
        m_hDataReady    = CreateEvent (NULL, false, false,
NULL);
        m_hReadyToSend  = CreateEvent (NULL, false, false,
NULL);
        m_hSendThread = CreateThread ( NULL, 0,
(LPTHREAD_START_ROUTINE)AgentConnection::SendProxy,
                (LPVOID) this, 0, &dwThreadID);
}

void AgentConnection::SendThread ( )
{
        DWORD dwStart, dwEnd;
        TCHAR strResult [100];
        // Loop for sending the video data
        while ( true )
        {
                SetEvent ( m_hReadyToSend );
                WaitForSingleObject ( m_hDataReady, WAIT_TIME );
                if (WaitForSingleObject (m_hSignal, 0) ==
WAIT_OBJECT_0) return;

                dwStart = GetTickCount ( );
                VideoSend ( (LPBYTE)m_pTxheader,
sizeof(InfoBlock));
                if ( InfoBlock::PALETTE_AVAIL & m_pTxheader-
>fCommands)
                {
                        VideoSend ( (LPBYTE)m_pTxPal,   256 * sizeof
(PALETTEENTRY));
```

```
            }
            if ( VIDEO_PAINT == m_pTxheader->fStatus )
            {
                VideoSend ( (LPBYTE)m_pTxDirtyArray,
m_pTxheader->nDirtyCount * sizeof (DirtyBlock));
                VideoSend ( m_pTxCompressedBlock,
m_pTxheader->cbCompressedSize);
            }
            dwEnd = GetTickCount ( );

            m_scheme.SaveSendTime ( dwEnd - dwStart );
            if (m_bLogResults)
            {
                wsprintf ( strResult, "Network IO iteration:
%lu \n", (dwEnd-dwStart) );
                TRACE (strResult);
            }
        }
}

////////////////////////////////////////////////////////////
////////////////////////////
// Input Loop

bool AgentConnection::InitInputLoop ( )
{
    if (InputWait ( ) == false)
    {
        Disconnect ( );
        return false;
    }
    TRACE ("Input loop Wait successful\n");
    if (InputHandshake ( ) == false)
    {
        Disconnect ( );
        return false;
    }
    TRACE ("Input loop Handshake successful\n");
    return true;
}

////////////////////////////////////////////////////////////
////////////////////
// Input Loop

// protocol for the input loop:  wait for 4 byte command,
the next chunk of data depends
```

```
// on the paticular command.  Create the event locally.

int AgentConnection::InputLoop ( )
{
     TRACE ("Entering InputLoop\n");
     if (InitInputLoop ( ) == false)
     {
          return INPUT_EXIT_HANDSHAKE_ERROR;
     }
     HANDLE hSignal = OpenEvent (EVENT_ALL_ACCESS, false,
AGENT_EVENT);
     DWORD dwCommand;
     KeyboardEvent k_event;
     MouseEvent m_event, m_down, m_up;
     int n;

     bool bContinue = true;
     while (bContinue)
     {
          InputRecv ((LPBYTE)&dwCommand, sizeof
(dwCommand));

          HDESK hDesk = OpenInputDesktop ( 0, false,
DESKTOP_WRITEOBJECTS );
          if (NULL == hDesk) LAST_ERROR ( );
          else
          {
               BOOL_CALL ( SetThreadDesktop ( hDesk ) );
          }

          switch (dwCommand)
          {
          case INPUT_DOUBLE_CLICK_MOUSE:
               InputRecv ((LPBYTE)&m_down, sizeof
(m_event));
               InputRecv ((LPBYTE)&m_up, sizeof (m_event));
               for (n = 0; n < 2; n++)
               {
                    mouse_event (m_down.dwFlags, m_down.dx,
m_down.dy, m_down.dwData,
                         m_down.dwExtraInfo);
                    mouse_event (m_up.dwFlags, m_up.dx,
m_up.dy, m_up.dwData,
                         m_up.dwExtraInfo);
               }
               break;
          case INPUT_MOUSE:
```

```
                InputRecv ((LPBYTE)&m_event, sizeof
(m_event));
                mouse_event (m_event.dwFlags, m_event.dx,
m_event.dy, m_event.dwData,
                     m_event.dwExtraInfo);
                break;
           case INPUT_KEYBOARD:
                InputRecv ((LPBYTE)&k_event, sizeof
(k_event));
                keybd_event (k_event.Vk, k_event.Scan,
k_event.dwFlags, k_event.dwExtraInfo);
                break;
           case INPUT_PAINTING_PAUSE:
                InterlockedExchange ( &m_fStatus,
CONNECTION_PAUSE );
                break;
           case INPUT_PAINTING_RESUME:
                InterlockedExchange ( &m_fStatus,
CONNECTION_TRANSFER );
                break;
           case INPUT_CLOSE_CONNECTION:
                SetEvent (hSignal);
                bContinue = false;
                break;
           case INPUT_HOTKEY:
                break;
           default:
                TRACE ( "Invalid input command\n");
           }
           // check and see if the video loop is closing
           if (WaitForSingleObject (hSignal, 0) ==
WAIT_OBJECT_0)
           {
                TRACE ("Input event signaled\n");
                bContinue = false;
           }
      }
      CloseHandle (hSignal);
      // Update Interface
      SetWindowText (GetDlgItem (m_hDlg, IDC_STATUS),
AGENT_UI_WAITING);
      SetWindowText (GetDlgItem (m_hDlg, IDC_WHO), _T(""));
      SetWindowText (GetDlgItem (m_hDlg, IDC_IP ), _T(""));
      PostMessage (GetDlgItem (m_hDlg, IDC_INPUT),
STM_SETIMAGE, ICON_BIG,
                (WPARAM)m_hIconWait);
      // end Update
```

```
     TRACE ("Exiting Input Thread.");
     return INPUT_EXIT_SUCCESS;
}

////////////////////////////////////////////////////////////
/

void AgentConnection::StartThreads ( )
{
     DWORD dwThreadID;
     CreateThread ( NULL, 0,
(LPTHREAD_START_ROUTINE)AgentConnection::MonitorLoopProxy,
          (LPVOID)this, 0, &dwThreadID);
}

int AgentConnection::MonitorLoop ( )
{
     TRACE ("Entering Monitor loop.\n");
     DWORD dwThreadID;
     DWORD dwInputCode;
     DWORD dwVideoCode;

     HANDLE hThread[3];
     if (InputListen ( ) == false || VideoListen ( ) ==
false)
     {
          // MessageBox (NULL, "Unable to initialize
network.", "E-Parcel SmartConsultant", MB_OK |
MB_TASKMODAL);
          return -1;
     }
     // ** user interface stuff - must remove

     TCHAR Name [STATIC_BUFFER];
     int len = STATIC_BUFFER;
     if (m_ListenVideo.ServerName (Name, len))
          SetWindowText (GetDlgItem (m_hDlg,
IDC_CLIENT_NAME), Name);
     else SetWindowText (GetDlgItem (m_hDlg,
IDC_CLIENT_NAME), "Unkown");
     len = STATIC_BUFFER;
     if (m_ListenVideo.ServerIP (Name, len))
          SetWindowText (GetDlgItem (m_hDlg,
IDC_CLIENT_IP),   Name);
     // ** end ui
     //while (true)
     {
```

```
          SendMessage (GetDlgItem (m_hDlg, IDC_INPUT),
STM_SETIMAGE, ICON_BIG,
               (WPARAM)m_hIconOff);
          ResetEvent (m_hSignal);
          hThread[0] = CreateThread ( NULL, 0,
(LPTHREAD_START_ROUTINE)InputLoopProxy,
               (LPVOID)this, 0, &dwThreadID);
          hThread[1] = CreateThread ( NULL, 0,
(LPTHREAD_START_ROUTINE)VideoLoopProxy,
               (LPVOID)this, 0, &dwThreadID);
          hThread[2] = OpenEvent (EVENT_ALL_ACCESS, false,
AGENT_EVENT);

          WaitForMultipleObjects (3, hThread, false,
INFINITE);
          if ( WaitForMultipleObjects (2, hThread, true,
2000) == WAIT_TIMEOUT)
          {
               TerminateThread ( hThread[0], 0xffff );
               TerminateThread ( hThread[1], 0xffff );
               TRACE ("*** Terminating threads ***\n");
          }

          GetExitCodeThread ( hThread[0], &dwInputCode);
          GetExitCodeThread ( hThread[1], &dwVideoCode);
          CloseHandle (hThread[0]);
          CloseHandle (hThread[1]);
          CloseHandle (hThread[2]);

          if (VIDEO_EXIT_HARDWARE_ERROR      ==
dwVideoCode)
          {
               TRACE ( "Unable to enter the proper video
mode \n" );
          //   MessageBox (m_hDlg, "Unable to enter the
proper video mode.", "E-Parcel SmartConsultant",
          //        MB_OK | MB_TASKMODAL);
          }
     }
     PostMessage ( m_hDlg, WM_DESTROY, 0, 0 );
     return 0;
}


void AgentConnection::SetCompressionAlgorithm ( int nId )
{
     switch (nId)
```

```
        {
        case IDC_1:
             m_fCompressionAlgorithm = CPX_CUSTOM_RLE;
             TRACE ("Switching to custom RLE\n");
             break;
        case IDC_2:
             m_fCompressionAlgorithm = CPX_HUFFMAN_RLE;
             TRACE ("Switching to huffman w/RLE\n");
             break;
        case IDC_3:
             m_fCompressionAlgorithm = CPX_CRUSHER_RLE_9;
             TRACE ("Switching to Crusher 9/rle\n");
             break;
        case IDC_4:
             m_fCompressionAlgorithm = CPX_CRUSHER_RLE_13;
             TRACE ("Switching to Crusher 13/rle\n");
             break;
        }
}

void AgentConnection::UpdateCompressionUI ( )
{
        PostMessage (GetDlgItem (m_hDlg, IDC_1), BM_SETCHECK,
0, 0);
        PostMessage (GetDlgItem (m_hDlg, IDC_2), BM_SETCHECK,
0, 0);
        PostMessage (GetDlgItem (m_hDlg, IDC_3), BM_SETCHECK,
0, 0);
        PostMessage (GetDlgItem (m_hDlg, IDC_4), BM_SETCHECK,
0, 0);
        switch (m_fCompressionAlgorithm)
        {
        case CPX_CUSTOM_RLE:
             PostMessage (GetDlgItem (m_hDlg, IDC_1),
BM_SETCHECK, 1, 0);
             TRACE ("Switching to custom RLE\n");
             break;
        case CPX_HUFFMAN_RLE:
             PostMessage (GetDlgItem (m_hDlg, IDC_2),
BM_SETCHECK, 1, 0);
             TRACE ("Switching to huffman w/RLE\n");
             break;
        case CPX_CRUSHER_RLE_9:
             PostMessage (GetDlgItem (m_hDlg, IDC_3),
BM_SETCHECK, 1, 0);
             TRACE ("Switching to Crusher 9/rle\n");
             break;
```

```
     case CPX_CRUSHER_RLE_13:
          PostMessage (GetDlgItem (m_hDlg, IDC_4),
BM_SETCHECK, 1, 0);
          TRACE ("Switching to Crusher 13/rle\n");
          break;
     default:
          ASSERT ( true );
          TRACE ("Invalid Compression Algorithm.\n");
          break;
     }
}

////////////////////////////////////////////////////////////
//
bool AgentConnection::InputListen ( )
{
     try
     {
          m_ListenInput.Create ( INPUT_PORT );
          SendMessage (GetDlgItem (m_hDlg, IDC_INPUT),
STM_SETIMAGE, ICON_BIG,
               (WPARAM)m_hIconOff);
          m_ListenInput.Listen ( );
     }
     catch (Except e)
     {
          e.Trace ( );
          TRACE ("Input Listen failed");
          return false;
     }
     TRACE ("Input Listen Success\n");
     return true;
}


////////////////////////////////////////////////////////////
//
// Sending and Receiving
bool AgentConnection::InputWait ( )
{
     try
     {
          m_ListenInput.Accept ( m_InputSocket );
          TRACE ("Input socket connected.\n");

          SendMessage (GetDlgItem   (m_hDlg, IDC_INPUT),
STM_SETIMAGE, ICON_BIG,
```

```
                (WPARAM)m_hIconWait);
            SetWindowText (GetDlgItem (m_hDlg, IDC_WHO),
"looking up name ...");
            SetWindowText (GetDlgItem (m_hDlg, IDC_STATUS),
AGENT_UI_CONNECTING);

            TCHAR strMsg  [STATIC_BUFFER * 4];
            TCHAR strHost [STATIC_BUFFER];
            TCHAR strIP   [STATIC_BUFFER];
            int len = STATIC_BUFFER;
            if (m_InputSocket.ClientName (strHost, len) ==
false)
            {
                wsprintf (strHost, "Unknown");
            }
            if (m_InputSocket.ClientIP (strIP, len) == false)
            {
                wsprintf (strIP, "Unknown");
            }
            m_Reject = false;
            if (SendMessage (GetDlgItem (m_hDlg,
IDC_ASK_PERMISSION), BM_GETCHECK, 0, 0) == BST_CHECKED)
            {
                wsprintf ( strMsg, _T("You have received a
request to connect from\r\n"
                                      "Host: %s\r\nfrom IP
address: %s\r\n"
                                      "Would you like to
accept?"), strHost, strIP);
                if (MessageBox (m_hDlg, strMsg, "Connection
Request", MB_YESNO | MB_ICONQUESTION) == IDNO)
                {
                    m_Reject = true;
                }
            }
            if (false == m_Reject)
            {
                // user interface update
                SetWindowText (GetDlgItem (m_hDlg,
IDC_STATUS), AGENT_UI_CONNECTED);
                SetWindowText (GetDlgItem (m_hDlg, IDC_WHO),
strHost);
                SetWindowText (GetDlgItem (m_hDlg, IDC_IP),
strIP);
                SendMessage (GetDlgItem (m_hDlg, IDC_INPUT),
STM_SETIMAGE, ICON_BIG,
                    (WPARAM)m_hIconOn);
```

```
                // end user interface
            }
            else
            {
                SendMessage (GetDlgItem (m_hDlg, IDC_INPUT),
STM_SETIMAGE, ICON_BIG,
                    (WPARAM)m_hIconWait);
            }
            HANDLE hSignal = OpenEvent (EVENT_ALL_ACCESS,
false, AGENT_ALLOW_EVENT);
            SetEvent (hSignal);
            CloseHandle (hSignal);
        }
        catch (Except e)
        {
            e.Trace ( );
            TRACE ("Input Wait Failed.\n");
            return false;
        }
        return true;
}


bool AgentConnection::VideoListen ( )
{
        try
        {
            m_ListenVideo.Create ( VIDEO_PORT );
            SendMessage (GetDlgItem (m_hDlg, IDC_VIDEO),
STM_SETIMAGE, ICON_BIG,
                (WPARAM)m_hIconOff);
            m_ListenVideo.Listen ( );
        }
        catch (Except e)
        {
            e.Trace ( );
            TRACE ("Video Listen Failed.\n");
            return false;
        }
        TRACE ("Video Listen Success.\n");
        return true;
}

//////////////////////////////////////////////////////////////
//////////////

bool AgentConnection::VideoWait ( )
```

```
{
     try
     {
          m_ListenVideo.Accept ( m_VideoSocket );
          TRACE ("Video socket connected.\n");
          HANDLE hSignal = OpenEvent (SYNCHRONIZE, false,
AGENT_ALLOW_EVENT);
          WaitForSingleObject (hSignal, INFINITE);
          CloseHandle (hSignal);
     }
     catch (Except e)
     {
          e.Trace ( );
          return false;
     }
     return true;
}


bool AgentConnection::VideoHandshake ( )
{
     int AgentVersion = VERSION, AdminVersion = 0;

     if (m_Reject) AgentVersion = REJECT;
     VideoSend ( (LPBYTE)&AgentVersion, sizeof (int) );
     if (m_Reject == false)
          VideoRecv ( (LPBYTE)&AdminVersion, sizeof (int)
);
     if (AgentVersion != AdminVersion || m_Reject)
     {
          TRACE ("Video Handshake failed.\n");
          return false;
     }
     return true;
}

bool AgentConnection::InputHandshake ( )
{
     int AgentVersion = VERSION, AdminVersion = 0;

     if (m_Reject) AgentVersion = REJECT;
     InputSend ( (LPBYTE)&AgentVersion, sizeof (int) );
     InputRecv ( (LPBYTE)&AdminVersion, sizeof (int) );
     if (AgentVersion != AdminVersion || m_Reject)
     {
          TRACE ("input Handshake failed.\n");
```

```
            return false;
        }
        return true;
}

////////////////////////////////////////////////////////////////
/////////////////
// Sending

void AgentConnection::InputSend (LPBYTE pMsg, int len)
{
        try
        {
            m_InputSocket.SendFully (pMsg, len);
        }
        catch (Except e)
        {
            TRACE ("Input Socket Send Failed\n");
            e.Trace ( );
            Disconnect ( );
        }
}

void AgentConnection::InputRecv (LPBYTE pMsg, int len)
{
        try
        {
            m_InputSocket.RecvFully (pMsg, len);
        }
        catch (Except e)
        {
            TRACE ("Input Socket Recv Failed\n");
            e.Trace ( );
            Disconnect ( );
        }
}

void AgentConnection::VideoSend (LPBYTE pMsg, int len)
{
        try
        {
            m_VideoSocket.SendFully (pMsg, len);
        }
        catch (Except e)
        {
            TRACE ("Video Socket Send Failed\n");
            e.Trace ( );
```

```
            Disconnect ( );
        }
        catch ( ... )
        {
            TRACE ("Unknown exception\n");
        }
}

void AgentConnection::VideoRecv (LPBYTE pMsg, int len)
{
        try
        {
            m_VideoSocket.RecvFully (pMsg, len);
        }
        catch (Except e)
        {
            TRACE ("Video Socket Recv Failed\n");
            e.Trace ( );
            Disconnect ( );
        }
}

////////////////////////////////////////////////////////////
///////


void AgentConnection::Disconnect ( )
{
        HANDLE hSignal = OpenEvent (EVENT_ALL_ACCESS, false,
AGENT_EVENT);
        SetEvent (hSignal);
        CloseHandle (hSignal);

        m_VideoSocket.Close ( );
        m_InputSocket.Close ( );
}


#include <windows.h>
#include "rle.h"
#include "bitio.h"
#include "ahuff.h"

/*
 * This data structure is all that is needed to maintain an
adaptive
```

```
 * Huffman tree for both encoding and decoding.  The leaf
array is a
 * set of indices into the nodes that indicate which node
is the
 * parent of a symbol.  For example, to encode 'A', we
would find the
 * leaf node by way of leaf[ 'A' ].  The next_free_node
index is used
 * to tell which node is the next one in the array that can
be used.
 * Since nodes are allocated when characters are read in
for the first
 * time, this pointer keeps track of where we are in the
node array.
 * Finally, the array of nodes is the actual Huffman tree.
The child
 * index is either an index pointing to a pair of children,
or an
 * actual symbol value, depending on whether
'child_is_leaf' is true
 * or false.
 */


/*
 * The Tree used in this program is a global structure.
Under other
 * circumstances it could just as well be a dynamically
allocated
 * structure built when needed, since all routines here
take a TREE
 * pointer as an argument.
 */

AdaptHuffComp::AdaptHuffComp ( )
{
}

AdaptHuffComp::~AdaptHuffComp ( )
{
}



long AdaptHuffComp::CompressBuffer (LPBYTE pIn, LPBYTE
pOut, long cbSize, bool bRle/*=false*/)
```

```
{
     long cbRleSize = cbSize;
     long compressed_size = 0;
     LPBYTE pData;

     // run length encode before compression
     if (bRle)
     {
          pData = new BYTE[cbSize];
          cbRleSize = rle_compress (pIn, pData, cbSize);
          *((long*)pOut) = cbRleSize;
          pOut += sizeof (long);
          compressed_size += sizeof (long);
     }
     else pData = pIn;

     try
     {
          int c;
          BIT_MANIP* output = OpenOutput (pOut, cbRleSize);
          InitializeTree(  );
          Buffer input ( pData, cbRleSize );
          while ( input.End ( ) == false )
          {
               input.Get (c);
               EncodeSymbol( c, output );
               UpdateModel( c );
          }
          EncodeSymbol( END_OF_STREAM, output );
          compressed_size += CloseOutput ( output );
     }
     catch ( int )
     {
          if (bRle) delete pData;
          return -1;
     }
    if (bRle)  delete pData;
     return compressed_size;
}



bool AdaptHuffComp::ExpandBuffer ( LPBYTE pIn, LPBYTE pOut,
long cbCompressedSize,
                                        long cbFullSize, bool
bRle/*=false*/)
```

```
{
     long cbRleLen;
     LPBYTE pData;
     bool bResult = false;

     if (bRle)
     {
          cbRleLen = *((long*) pIn);
          pIn += sizeof (long);
          pData = new BYTE [cbFullSize];
          cbCompressedSize -= sizeof (long);
     }
     else pData = pOut;
     try
     {
          int c;

          BIT_MANIP* input = OpenInput (pIn,
cbCompressedSize);
          InitializeTree(  );
          Buffer output ( pData, cbFullSize );

          while ( input->block.End ( ) == false )
          {
               c = DecodeSymbol( input );
               output.Put ( c );
               UpdateModel( c );
          }
          bResult = true;
     }
     catch ( int )
     {
          bResult = false;
          if (bRle) delete pData;
     }
     if (bRle && bResult)
     {
          bResult = rle_expand ( pData, pOut, cbRleLen,
cbFullSize );
          delete pData;
     }
     return bResult;
}
```

```
/*
 * The Expansion routine looks very much like the
compression routine.
 * It first initializes the Huffman tree, using the same
routine as
 * the compressor did.  It then sits in a loop, decoding
characters and
 * updating the model until it reads in an END_OF_STREAM
symbol.  At
 * that point, it is time to quit.
 *
 * This routine will accept a single additional argument.
If the user
 * passes a "-d" argument, the function will dump out the
Huffman tree
 * to stdout when the program is complete.
 */


/*
 * When performing adaptive compression, the Huffman tree
starts out
 * very nearly empty.  The only two symbols present
initially are the
 * ESCAPE symbol and the END_OF_STREAM symbol.  The ESCAPE
symbol has to
 * be included so we can tell the expansion prog that we
are transmitting a
 * previously unseen symbol.  The END_OF_STREAM symbol is
here because
 * it is greater than eight bits, and our ESCAPE sequence
only allows for
 * eight bit symbols following the ESCAPE code.
 *
 * In addition to setting up the root node and its two
children, this
 * routine also initializes the leaf array.  The ESCAPE and
END_OF_STREAM
 * leaf elements are the only ones initially defined, the
rest of the leaf
 * elements are set to -1 to show that they aren't present
in the
 * Huffman tree yet.
 */

void AdaptHuffComp::InitializeTree(  )
{
```

```
    int i;

    m_tree.nodes[ ROOT_NODE ].child             = ROOT_NODE
+ 1;
    m_tree.nodes[ ROOT_NODE ].child_is_leaf     = FALSE;
    m_tree.nodes[ ROOT_NODE ].weight            = 2;
    m_tree.nodes[ ROOT_NODE ].parent            = -1;

    m_tree.nodes[ ROOT_NODE + 1 ].child         =
END_OF_STREAM;
    m_tree.nodes[ ROOT_NODE + 1 ].child_is_leaf = TRUE;
    m_tree.nodes[ ROOT_NODE + 1 ].weight        = 1;
    m_tree.nodes[ ROOT_NODE + 1 ].parent        =
ROOT_NODE;
    m_tree.leaf[ END_OF_STREAM ]                = ROOT_NODE
+ 1;

    m_tree.nodes[ ROOT_NODE + 2 ].child         = ESCAPE;
    m_tree.nodes[ ROOT_NODE + 2 ].child_is_leaf = TRUE;
    m_tree.nodes[ ROOT_NODE + 2 ].weight        = 1;
    m_tree.nodes[ ROOT_NODE + 2 ].parent        =
ROOT_NODE;
    m_tree.leaf[ ESCAPE ]                       = ROOT_NODE
+ 2;

    m_tree.next_free_node                       = ROOT_NODE
+ 3;

    for ( i = 0 ; i < END_OF_STREAM ; i++ )
        m_tree.leaf[ i ] = -1;
}


void AdaptHuffComp::EncodeSymbol( unsigned int c,
BIT_MANIP* output )
{
    unsigned long code;
    unsigned long current_bit;
    int code_size;
    int current_node;

    code = 0;
    current_bit = 1;
    code_size = 0;
    current_node = m_tree.leaf[ c ];
    if ( current_node == -1 )
        current_node = m_tree.leaf[ ESCAPE ];
```

```
    while ( current_node != ROOT_NODE )
     {
        if ( ( current_node & 1 ) == 0 )
            code |= current_bit;
        current_bit <<= 1;
        code_size++;
        current_node = m_tree.nodes[ current_node ].parent;
    };
    OutputBits( output, code, code_size );
    if ( m_tree.leaf[ c ] == -1 )
     {
        OutputBits( output, (unsigned long) c, 8 );
        add_new_node( c );
    }
}

/*
 * Decoding symbols is easy.  We start at the root node,
then go down
 * the tree until we reach a leaf.  At each node, we decide
which
 * child to take based on the next input bit.  After
getting to the
 * leaf, we check to see if we read in the ESCAPE code.  If
we did,
 * it means that the next symbol is going to come through
in the next
 * eight bits, unencoded.  If that is the case, we read it
in here,
 * and add the new symbol to the table.
 */

int AdaptHuffComp::DecodeSymbol( BIT_MANIP* input )
{
    int current_node;
    int c;

    current_node = ROOT_NODE;
    while ( !m_tree.nodes[ current_node ].child_is_leaf )
     {
        current_node = m_tree.nodes[ current_node ].child;
        current_node += InputBit( input );
    }
    c = m_tree.nodes[ current_node ].child;
    if ( c == ESCAPE )
     {
        c = (int) InputBits( input, 8 );
```

```
        add_new_node( c );
    }
    return( c );
}

/*
 * UpdateModel is called to increment the count for a given
symbol.
 * After incrementing the symbol, this code has to work its
way up
 * through the parent nodes, incrementing each one of them.
That is
 * the easy part.  The hard part is that after incrementing
each
 * parent node, we have to check to see if it is now out of
the proper
 * order.  If it is, it has to be moved up the tree into
its proper
 * place.
 */
void AdaptHuffComp::UpdateModel( int c )
{
    int current_node;
    int new_node;

    if ( m_tree.nodes[ ROOT_NODE].weight == MAX_WEIGHT )
        RebuildTree( );
    current_node = m_tree.leaf[ c ];
    while ( current_node != -1 )
     {
        m_tree.nodes[ current_node ].weight++;
        for ( new_node = current_node ; new_node >
ROOT_NODE ; new_node-- )
            if ( m_tree.nodes[ new_node - 1 ].weight >=
                 m_tree.nodes[ current_node ].weight )
                break;
        if ( current_node != new_node )
          {
            swap_nodes( current_node, new_node );
            current_node = new_node;
        }
        current_node = m_tree.nodes[ current_node ].parent;
    }
}

/*
```

```
 * Rebuilding the tree takes place when the counts have
gone too
 * high.  From a simple point of view, rebuilding the tree
just means that
 * we divide every count by two.  Unfortunately, due to
truncation effects,
 * this means that the tree's shape might change.  Some
nodes might move
 * up due to cumulative increases, while others may move
down.
 */

void AdaptHuffComp::RebuildTree ( )
{
    int i;
    int j;
    int k;
    unsigned int weight;

    j = m_tree.next_free_node - 1;
    for ( i = j ; i >= ROOT_NODE ; i-- )
     {
        if ( m_tree.nodes[ i ].child_is_leaf )
          {
            m_tree.nodes[ j ] = m_tree.nodes[ i ];
            m_tree.nodes[ j ].weight = ( m_tree.nodes[ j
].weight + 1 ) / 2;
             j--;
        }
    }

/*
 * At this point, j points to the first free node.  I now
have all the
 * leaves defined, and need to start building the higher
nodes on the
 * tree. I will start adding the new internal nodes at j.
Every time
 * I add a new internal node to the top of the tree, I have
to check to
 * see where it really belongs in the tree.  It might stay
at the top,
 * but there is a good chance I might have to move it back
down.  If it
 * does have to go down, I use the memmove() function to
scoot everyone
```

```
 * bigger up by one node.  Note that memmove() may have to
be change
 * to memcpy() on some UNIX systems.  The parameters are
unchanged, as
 * memmove and  memcpy have the same set of parameters.
 */
    for ( i = m_tree.next_free_node - 2 ; j >= ROOT_NODE ;
i -= 2, j-- )
     {
        k = i + 1;
        m_tree.nodes[ j ].weight = m_tree.nodes[ i ].weight
+
                                   m_tree.nodes[ k ].weight;
        weight = m_tree.nodes[ j ].weight;
        m_tree.nodes[ j ].child_is_leaf = FALSE;
        for ( k = j + 1 ; weight < m_tree.nodes[ k ].weight
; k++ )
            ;
        k--;
        memmove( &m_tree.nodes[ j ], &m_tree.nodes[ j + 1
],
               ( k - j ) * sizeof( struct Tree::Node ) );
        m_tree.nodes[ k ].weight = weight;
        m_tree.nodes[ k ].child = i;
        m_tree.nodes[ k ].child_is_leaf = FALSE;
    }
/*
 * The final step in tree reconstruction is to go through
and set up
 * all of the leaf and parent members.  This can be safely
done now
 * that every node is in its final position in the tree.
 */
    for ( i = m_tree.next_free_node - 1 ; i >= ROOT_NODE ;
i-- )
     {
        if ( m_tree.nodes[ i ].child_is_leaf )
          {
            k = m_tree.nodes[ i ].child;
            m_tree.leaf[ k ] = i;
        }
          else
          {
            k = m_tree.nodes[ i ].child;
            m_tree.nodes[ k ].parent = m_tree.nodes[ k + 1
].parent = i;
        }
```

```
    }
}

/*
 * Swapping nodes takes place when a node has grown too big
for its
 * spot in the tree.  When swapping nodes i and j, we
rearrange the
 * tree by exchanging the children under i with the
children under j.
 */

void AdaptHuffComp::swap_nodes( int i, int j )
{
    struct Tree::Node temp;

    if ( m_tree.nodes[ i ].child_is_leaf )
        m_tree.leaf[ m_tree.nodes[ i ].child ] = j;
    else
     {
        m_tree.nodes[ m_tree.nodes[ i ].child ].parent = j;
        m_tree.nodes[ m_tree.nodes[ i ].child + 1 ].parent
= j;
    }
    if ( m_tree.nodes[ j ].child_is_leaf )
        m_tree.leaf[ m_tree.nodes[ j ].child ] = i;
    else
     {
        m_tree.nodes[ m_tree.nodes[ j ].child ].parent = i;
        m_tree.nodes[ m_tree.nodes[ j ].child + 1 ].parent
= i;
    }
    temp = m_tree.nodes[ i ];
    m_tree.nodes[ i ] = m_tree.nodes[ j ];
    m_tree.nodes[ i ].parent = temp.parent;
    temp.parent = m_tree.nodes[ j ].parent;
    m_tree.nodes[ j ] = temp;
}

/*
 * Adding a new node to the tree is pretty simple.  It is
just a matter
 * of splitting the lightest-weight node in the tree, which
is the highest
 * valued node.  We split it off into two new nodes, one of
which is the
```

```
 * one being added to the tree.  We assign the new node a
weight of 0,
 * so the tree doesn't have to be adjusted.  It will be
updated later when
 * the normal update process occurs.  Note that this code
assumes that
 * the lightest node has a leaf as a child.  If this is not
the case,
 * the tree would be broken.
 */
void AdaptHuffComp::add_new_node( int c )
{
    int lightest_node;
    int new_node;
    int zero_weight_node;

    lightest_node = m_tree.next_free_node - 1;
    new_node = m_tree.next_free_node;
    zero_weight_node = m_tree.next_free_node + 1;
    m_tree.next_free_node += 2;

    m_tree.nodes[ new_node ] = m_tree.nodes[ lightest_node
];
    m_tree.nodes[ new_node ].parent = lightest_node;
    m_tree.leaf[ m_tree.nodes[ new_node ].child ] =
new_node;

    m_tree.nodes[ lightest_node ].child         = new_node;
    m_tree.nodes[ lightest_node ].child_is_leaf = FALSE;

    m_tree.nodes[ zero_weight_node ].child          = c;
    m_tree.nodes[ zero_weight_node ].child_is_leaf  =
TRUE;
    m_tree.nodes[ zero_weight_node ].weight         = 0;
    m_tree.nodes[ zero_weight_node ].parent         =
lightest_node;
    m_tree.leaf[ c ] = zero_weight_node;
}

/*
 * All the code from here down is concerned with printing
the tree.
 * Printing the tree out is basically a process of walking
down through
 * all the nodes, with each new node to be printed getting
nudged over
```

```
 * far enough to make room for everything that has come
before.
 */



/************************** End of AHUFF.C
****************************/


/************************** Start of BITIO.C
************************
 *
 * This utility file contains all of the routines needed to
impement
 * bit oriented routines under either ANSI or K&R C.  It
needs to be
 * linked with every program used in the entire book.
 *
 */
#include <windows.h>

#include "bitio.h"

//////////////////////////////////////////////////////////

void Buffer::Put (const BYTE c)
{
     if ( pCur >= pEnd )
     {
          throw 0;
     }
     else
     {
          *pCur = c;
          pCur ++;
     }
}

void Buffer::Get (BYTE& c)
{
     if ( pCur >= pEnd )
     {
          throw 0;
     }
     else
     {
```

```
            c = *pCur;
            pCur ++;
        }
}

void Buffer::Get (int& c)
{
        if ( pCur >= pEnd )
        {
            throw 0;
        }
        else
        {
            c = *pCur;
            pCur ++;
        }
}

void Buffer::Get (unsigned int& c)
{
        if ( pCur >= pEnd )
        {
            throw 0;
        }
        else
        {
            c = *pCur;
            pCur ++;
        }
}

////////////////////////////////////////////////////////////

BIT_MANIP* OpenOutput ( LPBYTE pStartBlock, DWORD dwSize )
{
    BIT_MANIP *bit;
    bit = (BIT_MANIP *) new BIT_MANIP (pStartBlock,
dwSize);
    bit->rack = 0;
    bit->mask = 0x80;
    return( bit );
}

BIT_MANIP* OpenInput ( LPBYTE pStartBlock, DWORD dwSize )
{
    BIT_MANIP *bit;
```

```
    bit = (BIT_MANIP *) new BIT_MANIP (pStartBlock,
dwSize);
    bit->rack = 0;
    bit->mask = 0x80;
    return( bit );
}

long CloseOutput ( BIT_MANIP *bit_obj )
{
     if ( bit_obj->mask != 0x80 )
     {
          bit_obj->block.Put (bit_obj->rack);
     }
     long cbSize = bit_obj->block.CurLen ( );
    delete bit_obj;
     return cbSize;
}

void CloseInput ( BIT_MANIP *bit )
{
     delete bit;
}

void OutputBit ( BIT_MANIP * bit_obj, int bit )
{
    if ( bit )
        bit_obj->rack |= bit_obj->mask;
    bit_obj->mask >>= 1;
    if ( bit_obj->mask == 0 )
     {
          bit_obj->block.Put (bit_obj->rack);
          bit_obj->rack = 0;
          bit_obj->mask = 0x80;
    }
}

void OutputBits( BIT_MANIP *bit_obj, unsigned long code,
int count )
{
    unsigned long mask;
     static int temp_count = 0;
     static int fun_count  = 0;
     fun_count ++;

    mask = 1L << ( count - 1 );
    while ( mask != 0)
     {
```

```
        if ( mask & code )
            bit_obj->rack |= bit_obj->mask;
        bit_obj->mask >>= 1;
        if ( bit_obj->mask == 0 )
          {
              bit_obj->block.Put (bit_obj->rack);
              bit_obj->rack = 0;
              bit_obj->mask = 0x80;
              temp_count++;
        }
        mask >>= 1;
    }
}

int InputBit ( BIT_MANIP *bit_obj )
{
    int value;

    if ( bit_obj->mask == 0x80 )
     {
          // there was a check for end of file
        bit_obj->block.Get (bit_obj->rack);
    }
    value = bit_obj->rack & bit_obj->mask;
    bit_obj->mask >>= 1;
    if ( bit_obj->mask == 0 )
          bit_obj->mask = 0x80;
    return( value ? 1 : 0 );
}

unsigned long InputBits( BIT_MANIP *bit_obj, int bit_count
)
{
    unsigned long mask;
    unsigned long return_value;

    mask = 1L << ( bit_count - 1 );
    return_value = 0;
    while ( mask != 0)
     {
          if ( bit_obj->mask == 0x80 )
          {
              // there was a check for end of file
              bit_obj->block.Get (bit_obj->rack);
          }
          if ( bit_obj->rack & bit_obj->mask ) return_value
|= mask;
```

```
            mask >>= 1;
            bit_obj->mask >>= 1;
            if ( bit_obj->mask == 0 )  bit_obj->mask = 0x80;
        }
        return( return_value );
}

#include <windows.h>
#include <tchar.h>
#include <ddraw.h>
#include "consultant.h"
#include "crusher.h"
#include "gRect.h"
#include "rle.h"
#include "diag.h"
#include "bitio.h"
#include "huff.h"
#include "ahuff.h"
#include "compress.h"
#include "hardware.h"
#include "checksum.h"

// checksum.cpp
CheckSum::CheckSum ( )
{
     m_Width  = 0;
     m_Height = 0;
     //InitTable ( );
     //cx_lCRC32Polynomial = CX_CRC32_POLYNOMIAL ;
     m_ByteCount = 0;
}

CheckSum::~CheckSum ( )
{
     //ReleaseTable ( );
}

// must be called once before ComputeFullCheckSum
void CheckSum::Initialize  ( long BufferWidth, long
BufferHeight, long pitch, long ByteCount )
{
     TRACE ( "CheckSum Initialize ***\n");
     if ( ByteCount == 1 ) TRACE ( "ByteCount = 1\n");
     if ( ByteCount == 2 ) TRACE ( "ByteCount = 2\n");
     if ( ByteCount == 3 ) TRACE ( "ByteCount = 3\n");
     m_Width  = BufferWidth  * ByteCount;
     m_Height = BufferHeight * ByteCount;
```

```
    m_Pitch  = pitch;//      * ByteCount;
    m_ByteCount = ByteCount;

    m_LineLength = (m_Pitch)/ sizeof (DWORD);
    m_MaxLine    = m_Height / GRID_HEIGHT;
    m_First          = (m_Width / GRID_WIDTH) / sizeof
(DWORD);
    m_Second   = (m_Width / GRID_WIDTH) / sizeof (DWORD);
    if (((m_Height / GRID_WIDTH)  % sizeof(DWORD)) > 0)
        m_Second ++;
    m_Length     = (m_Height * m_Width) / sizeof (DWORD);

}


// walks the entire memory space and computes the checksum
for each location
// less overhead then the other version, does not need to
recompute each
// location
bool CheckSum::ComputeFullCheckSum ( LPDWORD pBlock )
{
    ASSERT ( m_Width != 0 && m_Height != 0 );

    DWORD dwRow = 0, dwCol = 0;
    DWORD dwCurCol = 0; DWORD dwCurLine = 0;
    ZeroMemory (m_dwCurrentCRC, sizeof (m_dwCurrentCRC));
    int n = 0;
    //int start = 0;
    if ( NULL == pBlock ) return false;
    LPDWORD pRow = pBlock;

    while (dwRow < GRID_HEIGHT)
    {
        for (dwCurLine = 0; dwCurLine < m_MaxLine;
dwCurLine ++)
        {
            // iterate each column
            dwCol = 0;
            pBlock = pRow;
            while (dwCol < GRID_WIDTH)
            {
                // do two at a time
                for (dwCurCol = 0; dwCurCol <= m_First;
dwCurCol += 1)
                {
```

```
                        m_dwCurrentCRC [dwCol][dwRow]+=
*(pBlock + dwCurCol) ^ n;
                        n ++;
                    }
                    pBlock += m_First;
                    dwCol ++;
                    for (dwCurCol = 0; dwCurCol <=
m_Second; dwCurCol += 1)
                    {
                        m_dwCurrentCRC [dwCol][dwRow]+=
*(pBlock + dwCurCol) ^ n;
                        n ++;
                    }
                    pBlock += m_Second;
                    dwCol ++;

                }
                pRow += m_LineLength;
                //start ++;
                //if (start >= CK_STEP) start = 0;
            }
            dwRow ++;
        }
        return true;
}


void CheckSum::InitTable ( )
{
    int     i ;
    int     j ;
    DWORD  lValue ;

     Ccitt32Table =  new DWORD [256];
     if ( Ccitt32Table )
     {
      for ( i = 0 ; i <= 255 ; i++ )
      {
          lValue = i ;
          for ( j = 8 ; j > 0 ; j-- )
          {
             if ( lValue & 1 )
                 lValue = ( lValue >> 1 ) ^
cx_lCRC32Polynomial ;
             else
                 lValue >>= 1 ;
          }
```

```
            Ccitt32Table[ i ] = lValue ;
        }
      }
}


void CheckSum::ReleaseTable()
{
   if ( Ccitt32Table )
      delete( Ccitt32Table ) ;
}



//////////////////////////////////////////////////////////////
////////////////////////////////////

// this version takes a pointer to the begining of memory,
and a rect describing the
// location to compute the checksum for
bool CheckSum::ComputeRectCheckSum ( LPDWORD pBlock, const
RECT& locRect, int x, int y )
{
     DWORD dwCheckSum = 0;
     // process each scan line within the rectangle
     if (NULL == pBlock) return false;
     static DWORD  dwLineLen     = (locRect.right -
locRect.left) * m_ByteCount;
     DWORD* pStartBlock   = (DWORD*)pBlock + ((locRect.top
* m_Pitch)/(sizeof(DWORD))) +

(locRect.left/sizeof(DWORD));

     DWORD *pBeginBlock, *pEndBlock;
     int nCurRow = locRect.top;
     int n = 0;
     while (nCurRow < locRect.bottom)
     {
          pBeginBlock = pStartBlock;
          pEndBlock   = (DWORD*)pStartBlock +
(dwLineLen/sizeof(DWORD));
          while (pStartBlock < pEndBlock)
          {
               dwCheckSum += *pStartBlock ^ n;
               pStartBlock += 1; //SCAN_STEP;
               n++;
          }
```

```
            pStartBlock = (pBeginBlock +=  m_Pitch /
sizeof(DWORD));
            nCurRow ++;
      }
      m_dwCurrentCRC[x][y] = dwCheckSum;
      return true;
}

#include <windows.h>
#include <tchar.h>
#include <ddraw.h>
#include "consultant.h"
#include "crusher.h"
#include "gRect.h"
#include "rle.h"
#include "diag.h"
#include "bitio.h"
#include "huff.h"
#include "ahuff.h"
#include "compress.h"
#include "hardware.h"
#include "checksum.h"
#include "clientvideo.h"


ClientVideo::ClientVideo ( )
{
      m_BitsPerPel = 0;
      m_ByteCount = 0;
      m_bSupportLocking = false;
}

ClientVideo::~ClientVideo ( )
{
      CloseSession ( );
}

bool ClientVideo::OpenSession (HWND  hWnd)
{
      if (//(ChangeSettings ( ) == false) ||
            (CollectInfo    ( ) == false)    )
            return false;

      m_BitsPerPel = m_ByteCount*8;
      // open the direct draw objects
      if ( m_display.Open ( m_ScreenWidth, m_ScreenHeight,
```

```
                                    m_OffscreenWidth,
m_OffscreenHeight,
                                    Video::SCREEN_CLIENT,
m_ByteCount, hWnd) == false)
          return false;
     m_bSupportLocking = m_display.SupportScreenLocking (
);

     if (m_bSupportLocking)
     {
          m_checksum.Initialize ( m_ScreenWidth,
m_ScreenHeight,
               m_display.GetSurfacePitch ( ), m_ByteCount);
     }
     else
     {
          m_checksum.Initialize ( m_OffscreenWidth,
m_OffscreenHeight,
               m_display.GetBufferPitch ( ), m_ByteCount);
     }

     m_hWnd = hWnd;

     // set up the parameters for the work to be done
     m_rctScreen          = Rect   ( m_ScreenWidth,
m_ScreenHeight, GRID_WIDTH, GRID_HEIGHT);
     m_rctOffscreen = Rect   ( (m_OffscreenWidth -
m_padding), m_OffscreenHeight,
          OFFSCREEN_WIDTH, (GRID_COUNT / OFFSCREEN_WIDTH));
     // m_cbTotalBufferSize  = (m_display.GetSurfacePitch (
) + m_padding) * m_ScreenHeight * (m_BitsPerPel / 8);
     m_cbRowBufferSize    = m_OffscreenWidth *
(m_ScreenHeight / GRID_HEIGHT) * m_ByteCount;
     m_cbTotalBufferSize  = m_cbRowBufferSize * (GRID_COUNT
/ OFFSCREEN_WIDTH);
     return true;
}

void ClientVideo::QueryHardware ( HardwareInfo& info )
{
     info.ScreenWidth  = m_ScreenWidth;
     info.ScreenHeight = m_ScreenHeight;
     info.ByteCount    = m_ByteCount;
     info.MaxGridCount = GridCount ( );
}
```

```
void ClientVideo::CloseSession ( )
{
     m_display.Close ( );
}

bool ClientVideo::CollectInfo ( )
{
     HDC hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
     if (hDC == NULL)
     {
          TRACE ("Unable to collect info.\n");
          return false;
     }
     m_ScreenWidth       = GetDeviceCaps (hDC, HORZRES);
     m_ScreenHeight      = GetDeviceCaps (hDC, VERTRES);
     m_ByteCount              = ( GetDeviceCaps (hDC,
BITSPIXEL) / BITS_BYTE );
     DeleteDC (hDC);

     m_padding = PADDING * (m_ScreenWidth /
PADDING_DIVISOR);
     m_OffscreenWidth  = ((m_ScreenWidth  / GRID_WIDTH) *
OFFSCREEN_WIDTH) + m_padding;
     m_OffscreenHeight = ( m_ScreenHeight / GRID_HEIGHT) *
(GRID_COUNT / OFFSCREEN_WIDTH);
     return true;
}

bool ClientVideo::ProcessFrame ( InfoBlock& header,
DirtyBlock* arrayDirty,
                                        const LPBYTE pComp,
DWORD fCommands)
{
     bool bResult = false;
     if (ProcessIteration ( header, arrayDirty, fCommands )
== true)
          bResult = CompressBuffer ( header, pComp );
     return bResult;
}


bool ClientVideo::ProcessIteration ( InfoBlock& header,
DirtyBlock* arrayDirty, DWORD fCommands )
{
     if (false == m_bSupportLocking )
          return ProcessIterationNoLock ( header,
arrayDirty, fCommands );
```

```
     header.Clear    ( );
     header.fStatus = VIDEO_NO_PAINT;
     LPBYTE pScreen;
     int nRowCount;
     if (m_display.GetScreenMemory ( m_rctScreen.FullArea (
), pScreen ) == false)
     {
          TRACE ("Unable to get video memory\n");
          if (false == m_display.RestoreLostSurface ( ))
               return false;
     }

     // why the SEH?  if any screen res change happens we
loose the surface memory
     bool bCkSum = false;
     __try{
          bCkSum = m_checksum.ComputeFullCheckSum (
(LPDWORD)pScreen );
          }
     __except ( 1 ) {
          TRACE ( "Checksum access violation\n" );
          bCkSum = false;
     }
     if (false == bCkSum) return false;

     if ( fCommands & FORCE_PAINT)
     {
          TRACE ("Paint forced.\n");
     }

     m_rctScreen.MoveFirst ( );
     m_rctOffscreen.MoveFirst ( );

     while (m_rctScreen.End ( ) == false)
          {
          if ( (fCommands & FORCE_PAINT) ||
                m_checksum.Dirty (m_rctScreen.GridPosX( ),
m_rctScreen.GridPosY( )) )
               {
               // the block has changed, blit it to the
offscreen surface
               // wait for the blitter???, (maybe change),
no transparency
               arrayDirty[header.nDirtyCount++].Mark
(m_rctScreen.GridPosX( ),
                    m_rctScreen.GridPosY( ));
```

```
                m_checksum.Synch (m_rctScreen.GridPosX( ),
m_rctScreen.GridPosY( ));
                m_display.GetScreenRect ( m_rctScreen,
m_rctOffscreen );
                m_rctOffscreen.MoveNext ( );
                }
           m_rctScreen.MoveNext ( );
           }
     nRowCount = (header.nDirtyCount / OFFSCREEN_WIDTH);
     if ((header.nDirtyCount % OFFSCREEN_WIDTH) > 0)
nRowCount++;
     header.cbFullSize  = nRowCount * m_cbRowBufferSize;
     // send the header
     if (0 == header.nDirtyCount)
     {
          return false;
     }
     // if we reach here we've built an offscreen buffer of
n dirty blocks
     return true;
}


bool ClientVideo::ProcessIterationNoLock ( InfoBlock&
header, DirtyBlock* arrayDirty, DWORD fCommands )
{
     header.Clear    ( );
     header.fStatus = VIDEO_NO_PAINT;
     LPBYTE pBuffer;
     int nRowCount;

     if (m_display.GetBufferMemory (
m_rctOffscreen.FullArea ( ), pBuffer ) == false)
     {
          TRACE ("Unable to get video memory\n");
          return false;
     }

     m_rctScreen.MoveFirst ( );
     m_rctOffscreen.MoveFirst ( );

     while (m_rctScreen.End ( ) == false)
          {
          m_display.GetScreenRect ( m_rctScreen,
m_rctOffscreen );
          m_checksum.ComputeRectCheckSum (
(LPDWORD)pBuffer, m_rctOffscreen,
```

```
                m_rctScreen.GridPosX( ),
m_rctScreen.GridPosY( ));
          if ( (fCommands & FORCE_PAINT) ||
                  m_checksum.Dirty (m_rctScreen.GridPosX( ),
m_rctScreen.GridPosY( )) )
                {
                // the block has changed, blit it to the
offscreen surface
                // wait for the blitter???, (maybe change),
no transparency
                arrayDirty[header.nDirtyCount++].Mark
(m_rctScreen.GridPosX( ),
                     m_rctScreen.GridPosY( ));
                m_checksum.Synch (m_rctScreen.GridPosX( ),
m_rctScreen.GridPosY( ));
                m_rctOffscreen.MoveNext ( );
                }
          m_rctScreen.MoveNext ( );
          }
     nRowCount = (header.nDirtyCount / OFFSCREEN_WIDTH);
     if ((header.nDirtyCount % OFFSCREEN_WIDTH) > 0)
nRowCount++;
     header.cbFullSize  = nRowCount * m_cbRowBufferSize;
     // send the header
     if (0 == header.nDirtyCount)
     {
          return false;
     }
     // if we reach here we've built an offscreen buffer of
n dirty blocks
     return true;
}


bool ClientVideo::CompressBuffer ( InfoBlock& header, const
LPBYTE pOut )
{
     // get the video buffer and compress
     LPBYTE pOffscreen;
     if (m_display.GetBufferMemory (
m_rctOffscreen.FullArea ( ), pOffscreen ) == false )
     {
          TRACE ("Unable to get buffer memory\n");
          return false;
     }
     if (m_compressionEngine.Compress ( pOffscreen,
          pOut, header.cbFullSize, header.cbCompressedSize,
```

```
            header.fCompression) == true)
        {
            header.fStatus = VIDEO_PAINT;
            return true;
        }
        else
        {
            TRACE ("Compression failed\n");
            return    false;
        }
}

bool ClientVideo::GetPalette ( InfoBlock& header,
LPPALETTEENTRY pPal )
{
        LPPALETTEENTRY pTempPal = NULL;
        int Count;
        if (m_display.GetEntries ( pTempPal, Count ) == true)
        {
            header.fCommands |= InfoBlock::PALETTE_AVAIL;
            CopyMemory ( pPal, pTempPal, sizeof
(PALETTEENTRY) * Count );
            return true;
        }
        else return false;
}

#include <windows.h>
#include "diag.h"
#include "socket.h"
#include "comm.h"

// Comm : shared Communication class for Admin and Client


// initialization

Comm::Comm ( HANDLE hSignal ) : m_hSignal ( hSignal )
{
        m_Connected = false;
        pVSock = NULL;
        pISock = NULL;
}

Comm::~Comm ( )
{
```

```
}

// client routines

bool Comm::Connect ( LPCTSTR pServer )
{
     bool bResult = false;
     try
     {
          m_ClVideoSocket.Connect (pServer, VIDEO_PORT);
          m_ClInputSocket.Connect (pServer, INPUT_PORT);
          pVSock = &m_ClVideoSocket;
          pISock = &m_ClInputSocket;
          bResult = true;
     }
     catch (Except e)
     {
          TRACE ("Connect Failed.\n");
          e.Trace ( );
     }

     m_Connected = bResult;
     return bResult;
}

// server routines

bool Comm::PrepareServer ( )
{
     bool bResult = false;
     try
     {
          m_ListenInput.Create ( INPUT_PORT );
          m_ListenVideo.Create ( VIDEO_PORT );
          m_ListenInput.Listen ( );
          m_ListenVideo.Listen ( );
          bResult = true;
     }
     catch (Except e)
     {
          TRACE ("Unable to Prepare Server");
          e.Trace ( );
     }
     return bResult;
}
```

```
bool Comm::Wait ( )
{
     bool bResult = false;
     try
     {
          m_ListenVideo.Accept ( m_SvVideoSocket );
          m_ListenInput.Accept ( m_SvInputSocket );
          pVSock = &m_SvVideoSocket;
          pISock = &m_SvInputSocket;
          bResult = true;
     }
     catch (Except e)
     {
          TRACE ("Unable to Wait");
          e.Trace ( );
     }

     m_Connected = bResult;
     return bResult;
}


bool Comm::RemoteInfo ( LPTSTR strHost, LPTSTR strIP, int
len )
{
     if (m_SvInputSocket.ClientName (strHost, len) ==
false)
     {
          wsprintf (strHost, "no entry");
     }
     if (m_SvInputSocket.ClientIP   (strIP, len) == false)
     {
          wsprintf (strIP,   "unknown");
     }
     return true;
}

// Close Routine

void Comm::Close ( )
{
     m_Connected = false;
     SetEvent ( m_hSignal );

     pISock->Close ( );
     pVSock->Close ( );
}
```

```
// IO routines

void Comm::InputSend (LPBYTE pMsg, int len)
{
     if (!m_Connected) return;
     try
     {
          pISock->SendFully (pMsg, len);
     }
     catch (Except e)
     {
          TRACE ("Input Socket Send Failed\n");
          e.Trace ( );
          Close ( );
     }
}

void Comm::InputRecv (LPBYTE pMsg, int len)
{
     if (!m_Connected) return;
     try
     {
          pISock->RecvFully (pMsg, len);
     }
     catch (Except e)
     {
          TRACE ("Input Socket Recv Failed\n");
          e.Trace ( );
          Close ( );
     }
}

void Comm::VideoSend (LPBYTE pMsg, int len)
{
     if (!m_Connected) return;
     try
     {
          pVSock->SendFully (pMsg, len);
     }
     catch (Except e)
     {
          TRACE ("Video Socket Send Failed\n");
          e.Trace ( );
          Close ( );
     }
```

```
}

void Comm::VideoRecv (LPBYTE pMsg, int len)
{
     if (!m_Connected) return;
     try
     {
          pVSock->RecvFully (pMsg, len);
     }
     catch (Except e)
     {
          TRACE ("Video Socket Recv Failed\n");
          e.Trace ( );
          Close ( );
     }
}


#include <windows.h>
#include "bitio.h"
#include "crusher.h"
#include "diag.h"
#include "rle.h"
#include "huff.h"
#include "ahuff.h"
#include "compress.h"


CompressionEngine::CompressionEngine ( )
{
     m_pRleBuffer = NULL;

     // initialize the compression algorithms
     if ( ccxVersion ( ) != CRUSHER_VERSION )
     {
          m_bFailCrusher = true;
          TRACE ("Unknown Crusher Version\n");
     }
     else
     {
          m_bFailCrusher = false;
          cxBuf2BufInit ( ); // crusher
     }
}

CompressionEngine::~CompressionEngine ( )
{
```

```
     // cleanup
     cxBuf2BufClose ( );
}

bool CompressionEngine::Compress ( LPBYTE pIn, LPBYTE pOut,
                                        const long
cbFullSize, long& cbCompressedSize,
                                        DWORD fMode )
{
     bool bResult = false;
     static DWORD fCachedMode = 0;
     if (fCachedMode != fMode)
     {
          if (fMode >= (DWORD)CPX_CRUSHER_RLE_9 && fMode <=
(DWORD)CPX_CRUSHER_RLE_13
               && false == m_bFailCrusher)
               ccxBuf2BufSetLevel ( ((short)fMode -
CPX_CRUSHER_RLE_9) + 9 );
     }
     fCachedMode = fMode;

#ifdef _USE_SEH_
     __try
     {
#endif
          cbCompressedSize = 0;
          switch (fMode)
          {
          case CPX_CUSTOM_RLE:
               cbCompressedSize   = rle_compress (pIn,
                    (LPBYTE)pOut, cbFullSize);
               if ( cbCompressedSize < 0 ) cbCompressedSize
= 0;
               else bResult = true;
               break;
          case CPX_HUFFMAN_RLE:
               cbCompressedSize   = m_huff.CompressBuffer
(pIn,
                    pOut, cbFullSize, true);
               if ( cbCompressedSize < 0 ) cbCompressedSize
= 0;
               else bResult = true;
               break;
          case CPX_ADAPT_HUFFMAN:
               cbCompressedSize   = m_Ahuff.CompressBuffer
(pIn,
                    pOut, cbFullSize, true);
```

```
                if ( cbCompressedSize < 0 ) cbCompressedSize
= 0;
                else bResult = true;
                break;
           /*
           case CPX_CRUSHER_12:
                cbCompressedSize    = cxBuf2BufCompress
((cxFPBUFFER)pIn,
                          (cxFPBUFFER)pOut, cbFullSize);
                if ( cbCompressedSize < 0 ) cbCompressedSize
= 0;
                else bResult = true;
                break;
           */
           case CPX_CRUSHER_RLE_9:
           case CPX_CRUSHER_RLE_10:
           case CPX_CRUSHER_RLE_11:
           case CPX_CRUSHER_RLE_12:
           case CPX_CRUSHER_RLE_13:
                cbCompressedSize = -1;
                if (RleCompressWrapStart (pIn, pOut,
cbFullSize) == true)
                {
                     cbCompressedSize    = cxBuf2BufCompress
((cxFPBUFFER)m_pRleBuffer,
                          (cxFPBUFFER)pOut, m_cbCompressed);
                     cbCompressedSize += sizeof(long);
                     RleCompressWrapFinish ( );
                }
                if ( cbCompressedSize < 0 ) cbCompressedSize
= 0;
                else bResult = true;
                break;
           default:
                TRACE("Unknown Compression Algorithm\n");
                break;
           }
#ifdef _USE_SEH_
      }
      __except ( true )
      {
           TRACE ("Access violation in the compression
routine\n");
           bResult = false;
      }
#endif
      return bResult;
```

```
}

bool CompressionEngine::Expand ( LPBYTE pIn, LPBYTE pOut,
                                                const long
cbFullSize, const long cbCompressedSize,
                                                DWORD fMode )
{
     bool bResult = false;
     static DWORD fCachedMode = 0;
     if (fCachedMode != fMode)
     {
          if (fMode >= (DWORD)CPX_CRUSHER_RLE_9 && fMode <=
(DWORD)CPX_CRUSHER_RLE_13)
               ccxBuf2BufSetLevel ( ((short)fMode -
CPX_CRUSHER_RLE_9) + 9 );
     }
     fCachedMode = fMode;

#ifdef _USE_SEH_
     __try
     {
#endif
          switch (fMode)
          {
          case CPX_CUSTOM_RLE:
               bResult = rle_expand (pIn, pOut,
cbCompressedSize, cbFullSize);
               break;
          case CPX_HUFFMAN_RLE:
               bResult = m_huff.ExpandBuffer  (pIn, pOut,
cbCompressedSize, cbFullSize, true);
               break;
          case CPX_ADAPT_HUFFMAN:
               bResult = m_Ahuff.ExpandBuffer (pIn, pOut,
cbCompressedSize, cbFullSize, true);
               break;
          /*
          case CPX_CRUSHER_12:
               bResult = (cxBuf2BufExpand ((cxFPBUFFER)pIn,
(cxFPBUFFER)pOut, cbFullSize,
                    cbCompressedSize) == CX_SUCCESS);
               break;
          */
          case CPX_CRUSHER_RLE_9:
          case CPX_CRUSHER_RLE_10:
          case CPX_CRUSHER_RLE_11:
          case CPX_CRUSHER_RLE_12:
```

```
          case CPX_CRUSHER_RLE_13:
               if (RleExpandWrapStart (pIn) == true)
               {
                    long cbComp = cbCompressedSize;
                    cbComp -= sizeof(long);
                    bResult = (cxBuf2BufExpand
((cxFPBUFFER)pIn, (cxFPBUFFER)m_pRleBuffer,
                         m_cbCompressed, cbComp) ==
CX_SUCCESS);
                    if (bResult)
                    {
                         bResult = RleExpandWrapFinish (
pOut, cbFullSize );
                    }
               }
               break;
          default:
               TRACE("Unknown Compression Algorithm\n");
               break;
          }
#ifdef _USE_SEH_
     }
     __except ( true )
     {
          TRACE ("Access violation in the decompression
routine.\n");
          bResult = false;
     }
#endif
     return bResult;
}



bool CompressionEngine::RleCompressWrapStart (LPBYTE pIn,
LPBYTE& pOut, const long cbSize)
{
     // run length encode before compression
     m_pRleBuffer = new BYTE[cbSize];
     if (m_pRleBuffer)
     {
          m_cbCompressed = rle_compress (pIn, m_pRleBuffer,
cbSize);
          *((long*)pOut) = m_cbCompressed;
          pOut += sizeof (long);
     }
```

```
    else return false;

    return (m_cbCompressed > 0);
}

bool CompressionEngine::RleCompressWrapFinish ( )
{
    if (m_pRleBuffer) delete m_pRleBuffer;
    return true;
}


bool CompressionEngine::RleExpandWrapStart (LPBYTE& pIn)
{
    m_cbCompressed = *((long*) pIn);
    pIn += sizeof (long);
    m_pRleBuffer = new BYTE [m_cbCompressed];
    if (m_pRleBuffer == NULL) return false;
    else return true;
}

bool CompressionEngine::RleExpandWrapFinish ( LPBYTE pOut,
long cbFullSize )
{
    bool bResult = rle_expand ( m_pRleBuffer, pOut,
m_cbCompressed, cbFullSize );
    delete m_pRleBuffer;

    return bResult;
}

// diagnostic functions
// writes to the debugger and (optionally) a log file
// by Rob Gagne 7/17/97

#include <windows.h>
#include <tchar.h>
#include "diag.h"


#ifdef _LOG_TRACE_
HANDLE hLogFile;
CRITICAL_SECTION cs;
void OpenLogFile (LPCTSTR strName)
{
    TCHAR dir [MAX_PATH];
    GetSystemDirectory (dir, MAX_PATH);
```

```
     TCHAR filename [255];
     wsprintf (filename, "%c:\\log.txt", dir[0]);
     hLogFile = CreateFile ( filename, GENERIC_WRITE,
FILE_SHARE_READ, NULL, CREATE_ALWAYS,
          0, NULL);
     TCHAR strLogLine [256];
     SYSTEMTIME time;
     DWORD dwWritten;
     wsprintf (strLogLine, "Log file for %s opened ",
strName);
     WriteFile (hLogFile, strLogLine, lstrlen (strLogLine),
          &dwWritten, NULL);
     GetLocalTime (&time);
     wsprintf (strLogLine, "%hu\\%hu %hu:%02hu\r\n",
          time.wMonth, time.wDay, time.wHour,
time.wMinute);
     WriteFile (hLogFile, strLogLine, lstrlen (strLogLine),
          &dwWritten, NULL);
     InitializeCriticalSection (&cs);
}
#endif


void Log_Trace (LPCTSTR pMsg)
{
     OutputDebugString (pMsg);
#ifdef _LOG_TRACE_
     DWORD dwWritten;
     EnterCriticalSection (&cs);
     WriteFile (hLogFile, pMsg, (lstrlen (pMsg)-1),
&dwWritten, NULL);
     WriteFile (hLogFile, "\r\n", 2, &dwWritten, NULL);

     // send to the trace window if it's there
     HWND hWnd = FindWindow ( NULL, "DiagWin" );
     COPYDATASTRUCT data;
     data.cbData = lstrlen ( pMsg ) + 5;
     data.lpData = (LPVOID)pMsg;
     SendMessage ( hWnd, WM_COPYDATA, (WPARAM)
GetCurrentProcessId( ), (LPARAM) &data );

     LeaveCriticalSection (&cs);

#endif
}

void Log_TraceLastError ( )
```

```
{
     DWORD dwError = GetLastError ( );
     TCHAR ErrorMsg [1000];
     wsprintf (ErrorMsg, _T("GetLastError = %lu\n"),
dwError);
     TRACE ( ErrorMsg );
     LPTSTR lpMsgBuf;
     if (FormatMessage(
          FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
          NULL, dwError,
          MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), //
Default language
          (LPTSTR) &lpMsgBuf, 0, NULL ) != false)
     {

          wsprintf (ErrorMsg, "Text = %s\n", lpMsgBuf);
          TRACE ( ErrorMsg );
          LocalFree( lpMsgBuf );
     }
}

bool DebugAssert (int nLine, LPTSTR strFile)
{
     TCHAR AssertMsg [1000];
     wsprintf (AssertMsg, "!!** Assertion **!! Line %d,
File %s\n", nLine, strFile);
     TRACE (AssertMsg);
     ExitProcess ( 0 );
     return 0;
}

// Agent.cpp


#include <windows.h>
#include <ddraw.h>
#include <tchar.h>
#include "crusher.h"
#include "consultant.h"
#include "resource.h"
#include "socket.h"
#include "rle.h"
#include "diag.h"
#include "bitio.h"
#include "huff.h"
#include "ahuff.h"
```

```
#include "compress.h"
#include "ratio.h"
#include "agent.h"
#include "gRect.h"
#include "hardware.h"
#include "checksum.h"
#include "clientvideo.h"
#include "systemsettings.h"

AgentConnection Agent;

int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hPrev,
                LPTSTR pComLine, int nComShow)
{
     OpenLogFile ("Client entered WinMain");
     InitializeInstance ( );

     SystemSettings system;
     if ( system.Set ( ) == false )
          return 0;

     Agent.CreateControlDialog ( );
     Agent.StartThreads ( );

     MSG msg;
     while (GetMessage (&msg, NULL, 0, 0))
     {
          if ( !IsDialogMessage (Agent.m_hDlg, &msg))
          {
               TranslateMessage (&msg);
               DispatchMessage (&msg);
          }
     }

     system.Restore ( );
     return 0;
}


BOOL CALLBACK AgentDlgProc (HWND hDlg, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
     switch (uMsg)
     {
     case WM_INITDIALOG:
          return true;
     case WM_COMMAND:
```

```
            DlgCommand ( hDlg, wParam, lParam);
            return true;
      case WM_PALETTECHANGED:
            Agent.SetPaletteChanged ( );
            return true;
      case WM_DISPLAYCHANGE:
            TRACE ("Display Resolution Change
Notification\n");
            return true;
      // deny suspend operations - only works on some
systems
      case WM_POWERBROADCAST:
            return BROADCAST_QUERY_DENY;
      case WM_CLOSE:
            if (MessageBox ( hDlg, "Are you sure you wish to
end the session?",
                  "Smart Consultant", MB_YESNO ) == IDNO )
return true;
      case WM_DESTROY:
            PostQuitMessage ( 0 );
            return true;
      default:
            return false;
      }
}

void DlgCommand (HWND hDlg, WPARAM wParam, LPARAM lParam)
{
      WORD wNotifyCode = HIWORD(wParam);
      WORD wID = LOWORD(wParam);
      HWND hwndCtl = (HWND) lParam;
     if (BN_CLICKED == wNotifyCode)
      {
            Agent.SetCompressionAlgorithm ( wID );
      }
      if (BN_CLICKED == wNotifyCode && IDC_LOG_RESULTS ==
wID)
      {
            Agent.SetLog ((SendMessage (GetDlgItem (hDlg,
                  IDC_LOG_RESULTS), BM_GETCHECK, 0, 0) ==
BST_CHECKED));
      }
      switch (wID)
      {
      case ID_STOP:
            break;
      case ID_MINIMIZE:
```

```
            ShowWindow ( hDlg, SW_MINIMIZE );
            break;
      case ID_SHUTDOWN:
            // Agent.RestoreSettings ( );
            if (MessageBox ( hDlg, "Are you sure you wish to
end the session?",
                  "Smart Consultant", MB_YESNO ) == IDNO )
return;
            EndDialog ( hDlg, 0 );
            PostQuitMessage ( 0 );
            break;
      }
}


/*
Instance Initialization: Winsock and kill the screen saver
*/

BOOL CALLBACK KillScreenSaverFunc(HWND hwnd, LPARAM
lParam);


void InitializeInstance ( )
{
      // start up the winsock stuff
      WSADATA ws;
      WSAStartup (0x0101, &ws);


      // kill the screen saver if it's running ( NT only )
      OSVERSIONINFO os = {0};
      os.dwOSVersionInfoSize = sizeof ( os );
      GetVersionEx ( &os );

      if ( VER_PLATFORM_WIN32_NT == os.dwPlatformId )
      {
            HDESK hdesk;
            hdesk = OpenDesktop(TEXT("Screen-saver"),
                                    0, FALSE,
                                    DESKTOP_READOBJECTS |
DESKTOP_WRITEOBJECTS);
            if (hdesk)
            {
                  EnumDesktopWindows(hdesk,
(WNDENUMPROC)KillScreenSaverFunc, 0);
                  CloseDesktop(hdesk);
```

```
            }
        }
}


BOOL CALLBACK KillScreenSaverFunc(HWND hwnd, LPARAM lParam)
{
  PostMessage(hwnd, WM_CLOSE, 0, 0);
  return TRUE;
}

#include <windows.h>
#include <tchar.h>
#include "socket.h"
#include "diag.h"

//////////////////////////////////////////////////////////////
///
//

Except::Except ( LPCTSTR pError ) : m_pError (pError)
{
     m_LastError = GetLastError ( );
}

void Except::Trace ( )
{
     DWORD dwError = GetLastError ( );
     TRACE ( m_pError );
     TRACE ( _T("\n") );
     TRACE ( _T("System Error: ") );
     LPTSTR lpMsgBuf;
     if (FormatMessage(
          FORMAT_MESSAGE_ALLOCATE_BUFFER |
FORMAT_MESSAGE_FROM_SYSTEM,
          NULL, dwError,
          MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), //
Default language
          (LPTSTR) &lpMsgBuf, 0, NULL ) != false)
     {
          TRACE ( lpMsgBuf );
          TRACE ( _T("\n") );
          LocalFree( lpMsgBuf );
     }
     else
     {
          TCHAR numBuf [20];
```

```
            wsprintf (numBuf, _T("%lu\n"), dwError);
            TRACE ( numBuf );
      }
}

#include <windows.h>
#include "gRect.h"


Rect::Rect (int Width, int Height, int Columns, int Rows)
      :           : m_nHeight(Height), m_nWidth(Width),
m_nRows(Rows), m_nColumns(Columns)
{
      m_GridW = m_nWidth  / m_nColumns;
      m_GridH = m_nHeight / m_nRows;
      left   = 0;
      top    = 0;
      bottom = m_GridH;
      right  = m_GridW;
      m_FullArea.top    = 0;
      m_FullArea.left   = 0;
      m_FullArea.right  = m_nWidth;
      m_FullArea.bottom = m_nHeight;
      m_GridArea = m_GridH * m_GridW;
}


RECT& Rect::MoveNext ( )
{
      m_x ++;
      if (m_x >= m_nColumns)
      {
            m_x = 0;
            m_y ++;
      }
      if (m_y >= m_nRows) m_bEnd = true;
      SetRect ( );
      return (*this);
}

RECT& Rect::MovePrev ( )
{
      m_x --;
      if (m_x <= 0)
      {
            m_x = (m_nColumns - 1);
            m_y --;
```

```
        }
        SetRect ( );
        return (*this);
}

RECT& Rect::MoveFirst ( )
{
        m_bEnd = false;
        m_x = 0; m_y = 0;
        SetRect ( );
        return (*this);
}


RECT& Rect::MoveTo (int x, int y)
{
        m_x = x;
        m_y = y;
        if (y > m_nRows) m_bEnd = true;
        SetRect ( );
        return (*this);
}


// hardware.cpp
// source file for the DirectDraw hardware abstraction
// July 25, 1997
// by Rob Gagne

#include <windows.h>
#include <tchar.h>
#include "consultant.h"
#include "ddraw.h"
#include "hardware.h"
#include "diag.h"


Video::Video ( )
{
        // data interface
        BitCount = 0;
        m_ByteCount = 0;

        // direct draw objects
        pDirectDraw = NULL;
        pScreen     = NULL;
        pOffscreen  = NULL;
```

```
    pPalette    = NULL;

    m_PalEntryCount = 0;
    m_pSavedEntries   = NULL;
    m_pCurrentEntries = NULL;
}

Video::~Video ( )
{
    Close ( );
}

// closing the objects
////////////////////////////////////////

void Video::Close ( )
{
    DD_CALL_INIT ( );
    if (pOffscreen)
    {
        DD_CALL (pOffscreen->Release ( ));
        pOffscreen = NULL;
    }
/*
    if (pPalette)
    {
        DD_CALL (pPalette->Release ( ));
        pPalette = NULL;
    }
*/
    if (pScreen)
    {
        DD_CALL (pScreen->Release ( ));
        pScreen = NULL;
    }
    if (pDirectDraw)
    {
        DD_CALL (pDirectDraw->RestoreDisplayMode ( ));
        DD_CALL (pDirectDraw->Release ( ));
        pDirectDraw = NULL;
    }
    if (m_pSavedEntries)
    {
        delete m_pSavedEntries;
        m_pSavedEntries = NULL;
    }
    if (m_pCurrentEntries)
```

```
    {
        delete m_pCurrentEntries;
        m_pCurrentEntries = NULL;
    }
}

bool Video::Open ( long w, long h, long off_w, long off_h,
DWORD fMode, int ByteCount,
                    HWND hWnd/*=NULL*/, LPPALETTEENTRY
pPal/*=NULL*/)
{
    ScreenWidth = w;
    ScreenHeight = h;
    OffscreenWidth = off_w;
    OffscreenHeight = off_h;
    m_ByteCount = ByteCount;
    BitCount = ByteCount * 8;

    m_hWnd = hWnd;
    switch (fMode)
    {
    case SCREEN_ADMIN:
        return OpenAdmin ( pPal );
    case SCREEN_CLIENT:
        return OpenClient ( );
    default:
        TRACE ("Bad Mode in Vido::Open\n");
        break;
    }
    return false;
}

////////////////////////////////////////////////////////////
////////////
// creating the direct draw objects

bool Video::OpenAdmin ( LPPALETTEENTRY pPal/*=NULL*/ )
{
    TRACE ( "** Opening Direct Draw Objects as Admin\n" );
    DD_CALL_INIT ( );

    // create direct draw object
    TRACE ("About to create DD object\n");
    DD_CALL (DirectDrawCreate (NULL, &pDirectDraw, NULL));
    if (DD_FAIL ( ))    return false;

    // set the cooperative level to exclusive
```

```
    TRACE ("About to set Coop Level\n");
    DD_CALL (pDirectDraw->SetCooperativeLevel (m_hWnd,
DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN ));
    if (DD_FAIL ( ))    return false;

    // change the resolution to match the client
    TRACE ("About to change display mode\n");
    DD_CALL (pDirectDraw->SetDisplayMode (
        ScreenWidth, ScreenHeight,    BitCount));
    if (DD_FAIL ( ))    return false;

    if ( BitCount == 8 )
    {    if (InitPaletteBuffers ( )    == false) return
false; }
    if (OpenPrimarySurface ( )    == false) return false;
    if (OpenBackBufferSurface ( ) == false) return false;
    if ( BitCount == 8 )
    {    if (OpenPalette ( pPal )   == false) return
false; }

    TRACE ( "** Direct Draw Objects Open\n" );
    return true;
}

bool Video::OpenClient ( )
{
    if ( BitCount != 8 ) return false;
    TRACE ( "** Opening Direct Draw Objects as Admin\n" );
    DD_CALL_INIT ( );
    // create direct draw object
    TRACE ( "Creating DD object \n");
    DD_CALL (DirectDrawCreate (NULL, &pDirectDraw, NULL));
    if (DD_FAIL ( ))    return false;

    // set the cooperative level to normal, we only want
to look at the screen
    TRACE ("Setting Coop Level\n");
    DD_CALL (pDirectDraw->SetCooperativeLevel (m_hWnd,
DDSCL_NORMAL));
    if (DD_FAIL ( ))    return false;

    if ( BitCount == 8 )
        if (InitPaletteBuffers ( )    == false) return
false;
    if (OpenPrimarySurface ( )    == false) return false;
    if (OpenBackBufferSurface ( ) == false) return false;
```

```
    TRACE ( "** Direct Draw Objects Open\n" );
    return true;
}

bool Video::OpenPrimarySurface ( )
{
    DD_CALL_INIT( );
    TRACE ("Opening primary surface\n");
    // create the surface
    DDSURFACEDESC dsc = {0};
    dsc.dwSize = sizeof (dsc);
    dsc.dwFlags = DDSD_CAPS;
    dsc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
    DD_CALL (pDirectDraw->CreateSurface (&dsc, &pScreen,
NULL));
    if (DD_FAIL ( ))    return false;

    // check to see if it supports surface locking
    // current implementation is to fail if it does not
    DDSURFACEDESC SurfaceDesc = {0};
    SurfaceDesc.dwSize = sizeof (SurfaceDesc);
    RECT rect;
    rect.left = rect.top = 0;
    rect.right = ScreenWidth;
    rect.bottom = ScreenHeight;
    TRACE ( "About to lock primary surface\n");
    DD_CALL (pScreen->Lock (&rect, &SurfaceDesc,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
    if (DD_FAIL ( ))
    {
        m_bSupportSLock = false;
        TRACE ("Screen does NOT support locking\n");
    }
    else
    {
        DD_CALL (pScreen->Unlock
(SurfaceDesc.lpSurface));
        m_bSupportSLock = true;
        TRACE ("Screen locking is supported\n");
    }
    return true;
}

bool Video::OpenBackBufferSurface ( )
{
    DD_CALL_INIT( );
    TRACE ("Opening Backbuffer\n");
```

```
    // Secondary Buffer for storing the dirty rectangles
    DDSURFACEDESC offdsc = {0};
    offdsc.dwSize = sizeof (offdsc);
    offdsc.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
    offdsc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
DDSCAPS_SYSTEMMEMORY;
    offdsc.dwHeight = OffscreenHeight;
    offdsc.dwWidth  = OffscreenWidth;
    DD_CALL (pDirectDraw->CreateSurface (&offdsc,
&pOffscreen, NULL));
    if (DD_FAIL ( ))    return false;

    // check to see if it supports surface locking
    // current implementation is to fail if it does not
    DDSURFACEDESC SurfaceDesc = {0};
    SurfaceDesc.dwSize = sizeof (SurfaceDesc);
    RECT rect;
    rect.left = rect.top = 0;
    rect.right  =  OffscreenWidth;
    rect.bottom =  OffscreenHeight;
    DD_CALL (pOffscreen->Lock (&rect, &SurfaceDesc,
         DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
    if (DD_FAIL ( ))
    {
         m_bSupportOLock = false;
         TRACE ("Offscreen Surface does NOT support
locking\n");
    }
    else
    {
         DD_CALL (pOffscreen->Unlock
(SurfaceDesc.lpSurface));
         m_bSupportOLock = true;
         TRACE ("Offscreen locking is supported\n");
    }

    // don't currently support non-locking surfaces
    // if (false == m_bSupportSLock) return false; give it
a try
    if (false == m_bSupportOLock) return false;

    return true;
}

// allocate data for holding the palette ( not the DD
object )
// for the client to keep track of palette changes
```

```
// rather then sending a new palette every iteration
bool Video::InitPaletteBuffers ( )
{
     if ( BitCount != 8 ) return true;

     m_pSavedEntries   = new PALETTEENTRY[MAX_PAL];
     m_pCurrentEntries = new PALETTEENTRY[MAX_PAL];
     if (m_pSavedEntries && m_pCurrentEntries)
m_PalEntryCount = MAX_PAL;
     return (m_pSavedEntries != NULL && m_pCurrentEntries
!= NULL);
}


// compare palettes, return true if they are the same
bool Video::CompareEntries ( LPPALETTEENTRY pEntries )
{
     if ( BitCount != 8 ) return true;

     for (int n = 0; n < MAX_PAL; n++)
     {
          if ( (m_pSavedEntries [n].peRed   !=
pEntries[n].peRed  )  ||
               (m_pSavedEntries [n].peBlue  !=
pEntries[n].peBlue )  ||
               (m_pSavedEntries [n].peGreen !=
pEntries[n].peGreen)  ||
               (m_pSavedEntries [n].peFlags !=
pEntries[n].peFlags) )
          {
               return false;
          }
     }
     return true;
}



// gets the direct draw object from the primary surface
// either takes an array of entries or creates one from the
// existing display if none are supplied
bool Video::OpenPalette ( LPPALETTEENTRY pEntries
/*=NULL*/)
{
     if ( BitCount != 8 ) return true;
     DD_CALL_INIT ( );
     if (pPalette)
```

```
    {
        DD_CALL (pPalette->Release ( ));
        pPalette = NULL;
    }
    if (pScreen)
    {
        TRACE ("Creating Palette\n");
        DD_CALL (pScreen->GetPalette ( &pPalette ));
        if (DD_FAIL ( ) )
        {
            if (NULL == pEntries)
            {
                HDC hDC = CreateDC ( _T("DISPLAY"),
NULL, NULL, NULL);
                ZeroMemory ( m_pSavedEntries, sizeof
(PALETTEENTRY) * MAX_PAL);
                GetSystemPaletteEntries ( hDC, 0,
MAX_PAL, m_pSavedEntries );
                DeleteDC ( hDC );
                pEntries = m_pSavedEntries;
            }

            DD_CALL (pDirectDraw->CreatePalette (
DDPCAPS_8BIT | DDPCAPS_ALLOW256,
                pEntries, &pPalette, NULL));
            if (pPalette)
            {
                TRACE ("About to set the palette\n");
                DD_CALL (pScreen->SetPalette ( pPalette
));
                if (DD_FAIL ( )) return false;
            }
        }
    }
    return ( pPalette != NULL );
}


// public interface call to get the entries
// fails if there are no changes
bool Video::GetEntries ( LPPALETTEENTRY& pEntries, int&
Count )
{
    if ( BitCount != 8 ) return true;
    HDC hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
    if (NULL == hDC) return false;
```

```
     UINT nColors = GetSystemPaletteEntries ( hDC, 0,
MAX_PAL, m_pSavedEntries );
     DeleteDC ( hDC );
     pEntries = m_pSavedEntries;
     Count = MAX_PAL;
     return true;
}

// sets the array of palette entries into the current
palette
bool Video::SetEntries ( const LPPALETTEENTRY pEntries, int
Count )
{
     if ( BitCount != 8 ) return true;
     DD_CALL_INIT ( );
     ASSERT (pPalette);
     if (pPalette)
     {
          DD_CALL (pPalette->SetEntries ( 0, 0, Count,
pEntries ));
          return DD_SUCCESS ( );
     }
     return false;
}

////////////////////////////////////////////////////////////
////////////////////////////
// Here lie the manipulation functions


// Blits a rect from the screen to a location in
// the offscreen buffer
bool Video::GetScreenRect ( RECT& scrn, RECT& offscrn )
{
     DD_CALL_INIT ( );
     DD_CALL (pOffscreen->BltFast (
          offscrn.left, offscrn.top,
          pScreen, &scrn,
          DDBLTFAST_WAIT | DDBLTFAST_NOCOLORKEY));
     return (DD_SUCCESS( ));
}

// Blits the rect from the offscreen surface to
// the screen
bool Video::PutScreenRect ( RECT& scrn, RECT& offscrn )
{
     DD_CALL_INIT ( );
```

```
    DD_CALL (pScreen->BltFast (
        scrn.left, scrn.top,
        pOffscreen, &offscrn,
        DDBLTFAST_WAIT | DDBLTFAST_NOCOLORKEY));
    return (DD_SUCCESS( ));
}


// surface locking / unlocking
bool Video::GetScreenMemory ( RECT* pRect, LPBYTE& pMem)
{
    ASSERT ( m_bSupportSLock );
    DD_CALL_INIT ( );
    DDSURFACEDESC SurfaceDesc = {0};
    SurfaceDesc.dwSize = sizeof (SurfaceDesc);
    DD_CALL (pScreen->Lock (pRect, &SurfaceDesc,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
    pMem = (LPBYTE)SurfaceDesc.lpSurface;
    DD_CALL (pScreen->Unlock (SurfaceDesc.lpSurface));
    return (pMem != NULL);
}

bool Video::GetBufferMemory ( RECT* pRect, LPBYTE& pMem )
{
    ASSERT ( m_bSupportOLock );
    DD_CALL_INIT ( );
    DDSURFACEDESC SurfaceDesc = {0};
    SurfaceDesc.dwSize = sizeof (SurfaceDesc);
    DD_CALL (pOffscreen->Lock (pRect, &SurfaceDesc,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL));
    pMem = (LPBYTE)SurfaceDesc.lpSurface;
    DD_CALL (pOffscreen->Unlock (SurfaceDesc.lpSurface));
    return (pMem != NULL);
}


// restore the surface
bool Video::RestoreLostSurface ( )
{
    DD_CALL_INIT ( );
    DD_CALL (pOffscreen->Restore ( ));
    DD_CALL (pScreen->Restore  ( ));
    return (DD_SUCCESS( ));
}

long Video::GetSurfacePitch ( )
{
```

```
	DD_CALL_INIT ( );
	if ( pScreen )
	{
		DDSURFACEDESC SurfaceDesc = {0};
		SurfaceDesc.dwSize = sizeof (SurfaceDesc);
		DD_CALL ( pScreen->GetSurfaceDesc ( &SurfaceDesc
) );
		return SurfaceDesc.lPitch;
	}
	return 0;
}

long Video::GetBufferPitch ( )
{
	DD_CALL_INIT ( );
	if ( pScreen )
	{
		DDSURFACEDESC SurfaceDesc = {0};
		SurfaceDesc.dwSize = sizeof (SurfaceDesc);
		DD_CALL ( pOffscreen->GetSurfaceDesc (
&SurfaceDesc ) );
		return SurfaceDesc.lPitch;
	}
	return 0;
}

// non-adaptve huffman encoding
// adapted from "The Data Compression Book 2nd edition"
// by Mark Nelson / Jean-Loup Gaily
// converted to buffer-buffer compression, c++ class
// by Rob Gagne 7/22/97

#include <windows.h>
#include "bitio.h"
#include "huff.h"
#include "rle.h"

HuffComp::HuffComp ( )
{
	CreateTables ( );
}

HuffComp::~HuffComp ( )
{
	CleanupTables ( );
}
```

```
void HuffComp::CreateTables ( )
{
     counts = new unsigned long [256];
     nodes  = new NODE [514];
     codes  = new CODE [257];
}

void HuffComp::CleanupTables ( )
{
     delete counts;
     delete nodes;
     delete codes;
}

void HuffComp::InitializeTables ( LPBYTE pIn, long cbSize)
{
    ZeroMemory (counts, sizeof(long) * 256);
     ZeroMemory (nodes,  sizeof(NODE) * 514);
     ZeroMemory (codes,  sizeof(CODE) * 257);
     count_bytes   ( pIn, cbSize, counts );
    scale_counts  ( counts, nodes );
}

long HuffComp::CompressBuffer (LPBYTE pIn, LPBYTE pOut,
long cbSize, bool bRle/*=false*/)
{
    int root_node;
     long cbRleSize = cbSize;
     long compressed_size = 0;
     LPBYTE pData;

     // run length encode before compression
     if (bRle)
     {
          pData = new BYTE[cbSize];
          cbRleSize = rle_compress (pIn, pData, cbSize);
          *((long*)pOut) = cbRleSize;
          pOut += sizeof (long);
          compressed_size += sizeof (long);
     }
     else pData = pIn;
     try
     {
          InitializeTables ( pData, cbRleSize );
          BIT_MANIP* output = OpenOutput (pOut, cbRleSize);
          // building the tree
```

```
            output_counts ( output, nodes );
            root_node = build_tree( nodes );
            convert_tree_to_code( nodes, codes, 0, 0,
root_node );
            // compression
            Buffer input ( pData, cbRleSize );
            compress_data( input, output, codes );
            compressed_size += CloseOutput ( output );
      }
      catch ( int )
      {
            return -1;
      }
     if (bRle)  delete pData;
      return compressed_size;
}



bool HuffComp::ExpandBuffer ( LPBYTE pIn, LPBYTE pOut, long
cbCompressedSize,
                                        long cbFullSize, bool
bRle/*=false*/)
{
      long cbRleLen;
      LPBYTE pData;
      ZeroMemory (nodes, sizeof(NODE)*514);
      bool bResult = false;

      if (bRle)
      {
            cbRleLen = *((long*) pIn);
            pIn += sizeof (long);
            pData = new BYTE [cbFullSize];
            cbCompressedSize -= sizeof (long);
      }
      else pData = pOut;
      try
      {
            // input the nodes
            BIT_MANIP* input = OpenInput (pIn,
cbCompressedSize);
            int root_node;
            input_counts( input, nodes );
            root_node = build_tree( nodes );
            // expansion
```

```
            Buffer output ( pData, cbFullSize );
            expand_data( input, output, nodes, root_node );
            bResult = true;
        }
        catch ( int )
        {
            bResult = false;
            if (bRle) delete pData;
        }
        if (bRle && bResult)
        {
            bResult = rle_expand ( pData, pOut, cbRleLen,
cbFullSize );
            delete pData;
        }
        return bResult;
}


 /*
  * In order for the compressor to build the same model, I
have to store
  * the symbol counts in the compressed file so the expander
can read
  * them in.  In order to save space, I don't save all 256
symbols
  * unconditionally.  The format used to store counts looks
like this:
  *
  *  start, stop, counts, start, stop, counts, ... 0
  *
  * This means that I store runs of counts, until all the
non-zero
  * counts have been stored.  At this time the list is
terminated by
  * storing a start value of 0.  Note that at least 1 run of
counts has
  * to be stored, so even if the first start value is 0, I
read it in.
  * It also means that even in an empty file that has no
counts, I have
  * to pass at least one count.
  *
  * In order to efficiently use this format, I have to
identify runs of
```

```
 * non-zero counts.  Because of the format used, I don't
want to stop a
 * run because of just one or two zeros in the count
stream.  So I have
 * to sit in a loop looking for strings of three or more
zero values in
 * a row.
 *
 * This is simple in concept, but it ends up being one of
the most
 * complicated routines in the whole program.  A routine
that just
 * writes out 256 values without attempting to optimize
would be much
 * simpler, but would hurt compression quite a bit on small
files.
 *
 */
void HuffComp::output_counts( BIT_MANIP* output, NODE*
nodes )
{
    int first;
    int last;
    int next;
    int i;

    first = 0;
    while ( first < 255 && nodes[ first ].count == 0 )
        first++;
/*
 * Each time I hit the start of the loop, I assume that
first is the
 * number for a run of non-zero values.  The rest of the
loop is
 * concerned with finding the value for last, which is the
end of the
 * run, and the value of next, which is the start of the
next run.
 * At the end of the loop, I assign next to first, so it
starts in on
 * the next run.
 */
    for ( ; first < 256 ; first = next )
     {
         last = first + 1;
         for ( ; ; )
         {
```

```
                for ( ; last < 256 ; last++ )
                     if ( nodes[ last ].count == 0 )
     break;
                last--;
                for ( next = last + 1; next < 256 ; next++ )
                     if ( nodes[ next ].count != 0 )
     break;
                if ( next > 255 )
     break;
                if ( ( next - last ) > 3 )
     break;
                last = next;
           }
           // Here is where I output first, last, and all
the counts in between.

           output->block.Put (first);
           output->block.Put (last);
           for ( i = first ; i <= last ; i++ )
           {
                output->block.Put (nodes [i].count);
           }
     }
      output->block.Put ( 0 );
}

/*
 * When expanding, I have to read in the same set of
counts.  This is
 * quite a bit easier that the process of writing them out,
since no
 * decision making needs to be done.  All I do is read in
first, check
 * to see if I am all done, and if not, read in last and a
string of
 * counts.
 */


void HuffComp::input_counts( BIT_MANIP* input, NODE* nodes
)
{
     int first;
     int last;
     int i;
      for ( i = 0 ; i < 256 ; i++ )
           nodes[ i ].count = 0;
```

```
     input->block.Get ( first );
     input->block.Get ( last );
    for ( ; ; )
     {
          for ( i = first ; i <= last ; i++ )
          {
               input->block.Get ( nodes [ i ].count );
          }
          input->block.Get ( first );
          if ( first == 0 ) break;
          input->block.Get ( last );
    }
    nodes[ END_OF_STREAM ].count = 1;
}


/*
 * This routine counts the frequency of occurence of every
byte in
 * the input file.  It marks the place in the input stream
where it
 * started, counts up all the bytes, then returns to the
place where
 * it started.  In most C implementations, the length of a
file
 * cannot exceed an unsigned long, so this routine should
always
 * work.
 */

void HuffComp::count_bytes( LPBYTE pIn, long cbLen,
unsigned long counts [])
{
    // int i;
     // clear the counter
    // for ( i = 0 ; i < 256 ; i++ ) counts[ i ] = 0;

     LPBYTE pEnd = pIn + cbLen;
    while ( pIn < pEnd )
     {
          counts[ (int)((BYTE)*pIn) ]++;
          pIn ++;
    }
}
```

```
/*
 * In order to limit the size of my Huffman codes to 16
bits, I scale
 * my counts down so they fit in an unsigned char, and then
store them
 * all as initial weights in my NODE array.  The only thing
to be
 * careful of is to make sure that a node with a non-zero
count doesn't
 * get scaled down to 0.  Nodes with values of 0 don't get
codes.
 */
void HuffComp::scale_counts( unsigned long* counts, NODE*
nodes )
{
    unsigned long max_count;
    int i;

    max_count = 0;
    for ( i = 0 ; i < 256 ; i++ )
       if ( counts[ i ] > max_count )
        max_count = counts[ i ];
    if ( max_count == 0 ) {
     counts[ 0 ] = 1;
     max_count = 1;
    }
    max_count = max_count / 255;
    max_count = max_count + 1;
    for ( i = 0 ; i < 256 ; i++ ) {
     nodes[ i ].count = (unsigned int) ( counts[ i ] /
max_count );
     if ( nodes[ i ].count == 0 && counts[ i ] != 0 )
         nodes[ i ].count = 1;
    }
    nodes[ END_OF_STREAM ].count = 1;
}

/*
 * Building the Huffman tree is fairly simple.  All of the
active nodes
 * are scanned in order to locate the two nodes with the
minimum
 * weights.  These two weights are added together and
assigned to a new
 * node.  The new node makes the two minimum nodes into its
0 child
```

```
 * and 1 child.  The two minimum nodes are then marked as
inactive.
 * This process repeats until their is only one node left,
which is the
 * root node.  The tree is done, and the root node is
passed back
 * to the calling routine.
 *
 * Node 513 is used here to arbitratily provide a node with
a guaranteed
 * maximum value.  It starts off being min_1 and min_2.
After all active
 * nodes have been scanned, I can tell if there is only one
active node
 * left by checking to see if min_1 is still 513.
 */
int HuffComp::build_tree( NODE* nodes )
{
    int next_free;
    int i;
    int min_1;
    int min_2;

    nodes[ 513 ].count = 0xffff;
    for ( next_free = END_OF_STREAM + 1 ; ; next_free++ ) {
     min_1 = 513;
     min_2 = 513;
     for ( i = 0 ; i < next_free ; i++ )
            if ( nodes[ i ].count != 0 ) {
                if ( nodes[ i ].count < nodes[ min_1
].count ) {
                    min_2 = min_1;
                    min_1 = i;
                } else if ( nodes[ i ].count < nodes[ min_2
].count )
                    min_2 = i;
            }
     if ( min_2 == 513 )
         break;
     nodes[ next_free ].count = nodes[ min_1 ].count
                              + nodes[ min_2 ].count;
        nodes[ min_1 ].saved_count = nodes[ min_1 ].count;
        nodes[ min_1 ].count = 0;
        nodes[ min_2 ].saved_count =  nodes[ min_2 ].count;
        nodes[ min_2 ].count = 0;
     nodes[ next_free ].child_0 = min_1;
     nodes[ next_free ].child_1 = min_2;
```

```
    }
    next_free--;
    nodes[ next_free ].saved_count = nodes[ next_free
].count;
    return( next_free );
}

/*
 * Since the Huffman tree is built as a decoding tree,
there is
 * no simple way to get the encoding values for each symbol
out of
 * it.  This routine recursively walks through the tree,
adding the
 * child bits to each code until it gets to a leaf.  When
it gets
 * to a leaf, it stores the code value in the CODE element,
and
 * returns.
 */

void HuffComp::convert_tree_to_code( NODE* nodes, CODE*
codes, unsigned int code_so_far, int bits, int node )
{
    if ( node <= END_OF_STREAM ) {
     codes[ node ].code = code_so_far;
     codes[ node ].code_bits = bits;
     return;
    }
    code_so_far <<= 1;
    bits++;
    convert_tree_to_code( nodes, codes, code_so_far, bits,
                          nodes[ node ].child_0 );
    convert_tree_to_code( nodes, codes, code_so_far | 1,
                          bits, nodes[ node ].child_1 );
}


void HuffComp::compress_data( Buffer& input, BIT_MANIP*
output, CODE* codes )
{
     int c;

     while (input.End ( ) == false)
     {
          input.Get ( c );
```

```
            OutputBits( output, (unsigned long) codes[ c
].code,
                    codes[ c ].code_bits );
        }
    OutputBits( output, (unsigned long) codes[
END_OF_STREAM ].code,
            codes[ END_OF_STREAM ].code_bits );
}

/*
 * Expanding compressed data is a little harder than the
compression
 * phase.  As each new symbol is decoded, the tree is
traversed,
 * starting at the root node, reading a bit in, and taking
either the
 * child_0 or child_1 path.  Eventually, the tree winds
down to a
 * leaf node, and the corresponding symbol is output.  If
the symbol
 * is the END_OF_STREAM symbol, it doesn't get written out,
and
 * instead the whole process terminates.
 */
void HuffComp::expand_data( BIT_MANIP* input, Buffer&
output, NODE* nodes, int root_node )
{
    int node;

    for ( ; ; )
     {
        node = root_node;
        do
          {
            if ( InputBit( input ) )
                node = nodes[ node ].child_1;
            else
                node = nodes[ node ].child_0;
          } while ( node > END_OF_STREAM );
     if ( node == END_OF_STREAM ) break;
     output.Put ( node );
    }
}
/*************************** End of HUFF.C
*************************/
```

```
// ratio.cpp

#include <windows.h>
#include "consultant.h"
#include "rle.h"
#include "diag.h"
#include "bitio.h"
#include "huff.h"
#include "ahuff.h"
#include "compress.h"
#include "ratio.h"


Ratio::Ratio ( )
{
     dwLastCollectionTime     = 0;
     dwLastTransmitionTime    = 0;
     dwCurrentCompression = MAX_COMPRESSION -1;

     flAvgRatio = 0;
     num = 0;

     // assign the comrpession levels
     arraySchemes [0] = CPX_CUSTOM_RLE;
     arraySchemes [1] = CPX_HUFFMAN_RLE;
     arraySchemes [2] = CPX_CRUSHER_RLE_9;
     arraySchemes [3] = CPX_CRUSHER_RLE_13;
}

Ratio::~Ratio ( )
{

}

DWORD Ratio::CompressionScheme ( )
{
     if ( (10 > dwLastCollectionTime) || (10 >
dwLastTransmitionTime) )
          return arraySchemes [dwCurrentCompression];

     float Ratio = ((float)dwLastCollectionTime /
(float)dwLastTransmitionTime);

     flAvgRatio = ((flAvgRatio * num) + Ratio) / (num + 1);
     num ++;
     if (num > MAX_NUM) num = MAX_NUM;
```

```
      // adjusts amount of compression
      if    (flAvgRatio >= MID_UPPER_LIMIT)
            dwCurrentCompression --;
      if    (flAvgRatio >= UPPER_LIMIT)
            dwCurrentCompression --;

      if (flAvgRatio <= MID_LOWER_LIMIT)
            dwCurrentCompression ++;
      if (flAvgRatio <= LOWER_LIMIT)
            dwCurrentCompression ++;

      // ensure it's in bounds
      if          (dwCurrentCompression < MIN_COMPRESSION)
            dwCurrentCompression = 0;
      else if (dwCurrentCompression >= MAX_COMPRESSION)
            dwCurrentCompression = (MAX_COMPRESSION -1);

      ////////////////////////////////////////////////
      // output ratios for diagnostic reasons
      /*
      TCHAR strRatio [250];
      TCHAR buf1 [50], buf2 [50];
      _gcvt ( (double)flAvgRatio, 4, buf1);
      _gcvt ( (double)Ratio, 4, buf2);
      wsprintf ( strRatio, "avg: %s, cur: %s\n", buf1,
buf2);
      TRACE (strRatio);
      */
      ////////////////////////////////////////////////
/

      return arraySchemes [dwCurrentCompression];
}

// buffer to buffer compression using run length encoding
// by Rob Gagne
// 7/21/97

#include <windows.h>
#include "rle.h"

const BYTE Marker  = 0xa1;



// compression format:
// byte
```

```
//  - or -
// marker, run-length, byte


bool rle_expand ( LPBYTE pIn, LPBYTE pOut, long
cbCompressedLen, long cbMaxOutput )
{
     BYTE curByte;
     unsigned char cbRun;
     LPBYTE pEnd = (pIn + cbCompressedLen);
     LPBYTE pOutputEnd = (pOut + cbMaxOutput);

     while ( (pIn < pEnd) && (pOut < pOutputEnd) )
     {
          curByte = *pIn;
          pIn ++;
          if (Marker == curByte)
          {
               cbRun   = *pIn;
               pIn ++;
               curByte = *pIn;
               pIn ++;
               while ( (cbRun > 0) && (pOut < pOutputEnd) )
               {
                    *pOut = curByte;
                    pOut ++;
                    cbRun --;
               }
          }
          else
          {
               *pOut = curByte;
               pOut ++;
          }
     }
     if (pIn == pEnd) return true;
     else return false;
}


long rle_compress ( LPBYTE pIn, LPBYTE pOut, long dwLen )
{
     LPBYTE pEnd = pIn + dwLen;
     LPBYTE pMaxOutput = pOut + dwLen;
     LPBYTE pOutStart  = pOut;
     BYTE CurByte;
```

```
    // make sure the last byte is something other then the
current
    BYTE LastByte = (*pIn) - 1;
    unsigned char cbRun = 1;

    while (pIn < (pEnd - 2) && pOut < pMaxOutput)
    {
        CurByte = *pIn;
        pIn++;
        if ( (CurByte == LastByte) ) cbRun ++;
        else
        {
            LastByte = CurByte;
            cbRun = 1;
        }
        // only want to encode runs of greater then 3
        // or the marker byte
        if ( (3 == cbRun) || (Marker == CurByte) )
        {
            if (cbRun == 3) pOut -= (cbRun -1);
            *pOut = Marker;
            pOut++;
            while ( (pIn < pEnd - 1)    && (pOut <
pMaxOutput) &&
                    (*pIn == CurByte) && (cbRun <
(BYTE)BYTE_MAX) )
            {
                pIn ++;
                cbRun ++;
            }
            *pOut = BYTE(cbRun);
            pOut++;
            *pOut = CurByte;
            pOut++;
            cbRun = 1;
            LastByte = (*pIn) - 1;
        }
        else
        {
            *pOut = CurByte;
            pOut++;
        }
    }
    return ((pOut >= pMaxOutput) ? -1 : (pOut -
pOutStart));
}
```

```
#include <windows.h>
#include <tchar.h>
#include "socket.h"
#include "consultant.h"
#include "diag.h"
////////////////////////////////////////////////////////
//
// socket classes


BaseSocket::BaseSocket ( )
{
     InitClass ( );
}

void BaseSocket::InitClass ( )
{
     m_socket = INVALID_SOCKET;
     ZeroMemory (&m_addr, sizeof (m_addr));
     m_nPort = 0;
     m_bCreated = false;
     m_bConnected = false;
}

BaseSocket::~BaseSocket ( )
{
     closesocket ( m_socket );
}

// locates the host
void BaseSocket::ResolveName ( int nPort, LPCTSTR Name )
{
     hostent* pHost = NULL;
     ZeroMemory (&m_addr, sizeof (m_addr));
     unsigned char NewName [4];
     // see if it is in dotted decimal form
     /*
     if ( IsIpAddr (Name, NewName))
     {
          // user entered the ip address directly
          CopyMemory (&m_addr.sin_addr, (const
char*)NewName, 4);
     }
     else
     {
          pHost = gethostbyname ( Name );
```

```
            // can't find it
            if ( !pHost )
            {
                 Except e ( _T("BaseSocket:: Resolving Host
Name") );
                 throw e;
            }
            CopyMemory (&m_addr.sin_addr, pHost->h_addr,
pHost->h_length);
            m_addr.sin_family = pHost->h_addrtype;
      }
      */

      // try to resolve
      pHost = gethostbyname ( Name );
      if ( pHost == NULL )
      {
            if ( IsIpAddr (Name, NewName))
            {
                 // user entered the ip address directly
                 CopyMemory (&m_addr.sin_addr, (const
char*)NewName, 4);
            }
            else
            {
                 Except e ( _T("BaseSocket:: Resolving Host
Name") );
                 throw e;
            }
      }
      else
      {
            CopyMemory (&m_addr.sin_addr, pHost->h_addr,
pHost->h_length);
            m_addr.sin_family = pHost->h_addrtype;
      }
      m_addr.sin_family = PF_INET;
      m_addr.sin_port = htons ((short)nPort);
}

// binds to the specified port and host name
void BaseSocket::Bind ( int nPort, LPCTSTR Name/*=NULL*/)
{
      ZeroMemory ( &m_addr, sizeof(m_addr));
      m_addr.sin_family = AF_INET;
      m_addr.sin_addr.s_addr = INADDR_ANY;
      m_addr.sin_port = htons ( nPort );
```

```
     if (bind (m_socket, (sockaddr*)&m_addr, sizeof
(m_addr)) == SOCKET_ERROR)
     {
          Except e ( _T("BaseSocket:: Binding to Host") );
          throw e;
     }
}

// Sending and Receiving

// returns number of bytes send or throws an error
int BaseSocket::Send ( LPBYTE pMsg, int nLen ) const
{
     int nSent;
     //if (CanWrite( ))
     nSent = send (m_socket, (char*)pMsg, nLen, 0);
     //else nSent = SOCKET_ERROR;
     if (SOCKET_ERROR == nSent)
     {
          Except e ( _T("BaseSocket:: Send Failure") );
          throw e;
     }
     return nSent;
}

// returns the number of bytes received or throws an error
int BaseSocket::Recv ( LPBYTE pMsg, int nLen ) const
{
     int nRecved;
     //if (CanRead ( ))
     nRecved = recv (m_socket, (char*)pMsg, nLen, 0);
     //else nRecved = SOCKET_ERROR;
     if (SOCKET_ERROR == nRecved)
     {
          Except e ( _T("BaseSocket:: Recv Failure") );
          throw e;
     }
     return nRecved;
}

bool BaseSocket::CanRead (int timeout /*=30*/ ) const
{
     timeval tout;
    tout.tv_sec = timeout;
    tout.tv_usec = 0;
    fd_set sockset;
```

```
    FD_ZERO(&sockset);
    FD_SET(m_socket, &sockset);
    return (select(m_socket+1, &sockset, NULL, NULL, &tout)
== 1);
}

bool BaseSocket::CanWrite (int timeout /*=30*/  ) const
{
     timeval tout;
    tout.tv_sec = timeout;
    tout.tv_usec = 0;
    fd_set sockset;
    FD_ZERO(&sockset);
    FD_SET(m_socket, &sockset);
    return (select(m_socket+1, NULL, &sockset, NULL, &tout)
== 1);
}

// returns number of bytes send or throws an error
int BaseSocket::SendFully ( LPBYTE pMsg, int nLen ) const
{
     int nSent = 0;
     //int CurSend;
     while (nSent < nLen)
     {
          //CurSend = ((nLen - nSent) > 0x10000) ? 0x10000
: (nLen - nSent);
          nSent += Send ( (pMsg + nSent), (nLen - nSent) );
     }
     return nSent;
}

// returns the number of bytes received or throws an error
int BaseSocket::RecvFully ( LPBYTE pMsg, int nLen ) const
{
     int nRecved = 0;
     //int CurRecv;
     while (nRecved < nLen)
     {
          //CurRecv = ((nLen - nRecved) > 0x10000) ?
0x10000 : (nLen - nRecved);
          nRecved += Recv ( (pMsg + nRecved), (nLen -
nRecved) );
     }
     return nRecved;
}
```

```
// empty incoming buffer
void BaseSocket::EmptyRecvBuffer ( ) const
{
     int nResult = 0;
     BYTE Buffer [100];
     try
     {
          while (nResult != SOCKET_ERROR && nResult != 0)
          {
               if (CanRead ( 1 ))
               {
                    nResult = Recv ( Buffer, sizeof
(Buffer));
               }
               else nResult = 0;
          }
     }
     catch (Except e)
     {
          OutputDebugString ("Empty Receive Buffer:\n");
          e.Trace ( );
          // all done
     }
}


void BaseSocket::Create ( )
{
     m_socket = socket (PF_INET, SOCK_STREAM, 0);
     if (INVALID_SOCKET == m_socket)
     {
          Except e ( _T("BaseSocket:: Create Failure\n") );
          throw e;
     }
     m_bCreated = true;
}

void BaseSocket::Shutdown (int nHow/*=SD_SEND*/)
{
     shutdown (m_socket, nHow);
}

void BaseSocket::Close ( )
{
     if (m_bConnected)
     {
          Shutdown ( );
```

```
            EmptyRecvBuffer ( );
            m_bConnected = false;
        }
        if (m_bCreated)
        {
            closesocket (m_socket);
            m_socket = INVALID_SOCKET;
            m_bCreated = false;
        }
}

bool BaseSocket::IsIpAddr ( LPCTSTR pAddr, unsigned char*
pDotted )
{
        TCHAR DottedForm [4][4] = {0};
        unsigned short nAddr [4];
        TCHAR* pToken;
        int n = 0;
        TCHAR pStr [STATIC_BUFFER];
        wsprintf ( pStr, _T("%s"), pAddr);
        pToken = strtok (pStr, _T(" ."));
        if (pToken) wsprintf (DottedForm[n], "%s", pToken);
        while (n < 4 && pToken)
        {
            n++;
            pToken = strtok (NULL, _T(" ."));
            if (pToken) wsprintf (DottedForm[n], "%s",
pToken);
        }
        for (n = 0; n < 4; n++)
        {
            nAddr [n] = atoi ( DottedForm [n] );
        }
        if ((nAddr [0] + nAddr [1] + nAddr [2] + nAddr [3]) ==
0)
            return false;
        for (n = 0; n < 4; n++)
        {
            pDotted [n] = (unsigned char) nAddr[n];
        }
        return true;
}

bool BaseSocket::IPFromAddr ( sockaddr_in* pAddr, LPTSTR
HostIP, int& nLen )
{
        if (nLen < (sizeof(TCHAR)*16))
```

```
    {
        nLen = (sizeof(TCHAR)*16);
        return false;
    }
    unsigned char* IP = (unsigned char*)&pAddr->sin_addr;
    unsigned short IPnum [4];
    IPnum[0] = (unsigned short) IP[0];
    IPnum[1] = (unsigned short) IP[1];
    IPnum[2] = (unsigned short) IP[2];
    IPnum[3] = (unsigned short) IP[3];
    wsprintf (HostIP, _T("%hu.%hu.%hu.%hu"), IPnum[0],
IPnum[1], IPnum[2], IPnum[3]);
    return true;
}

bool BaseSocket::NameFromAddr ( sockaddr_in* pAddr, LPTSTR
Host, int& nLen )
{
    hostent* pHost = gethostbyaddr ((const char*)&pAddr-
>sin_addr, 4, PF_INET);
    if (NULL == pHost)   return false;
    if (lstrlen (pHost->h_name)+1 > nLen)
    {
        nLen = lstrlen (pHost->h_name)+1;
        return false;
    }
    lstrcpy (Host, pHost->h_name);
    CharLower ( Host );
    return true;
}

////////////////////////////////////////////////////////////
//////////////////
// Server Socket

ServerSocket::ServerSocket (  )
{

}

void ServerSocket::Create ( int nPort )
{
    BaseSocket::m_nPort = nPort;
    BaseSocket::Create ( );
}

// blocking call
```

```
void ServerSocket::Accept ( ServerSocket& NewSocket )
{
     int addr_len = sizeof (m_client_addr);
     SOCKET c = accept (m_socket,
(sockaddr*)&m_client_addr, &addr_len);
     if ( INVALID_SOCKET == c )
     {
          Except e ( _T("SocketServer:: invalid socket
returned from accept\n") );
          throw e;
     }
     NewSocket = (*this);
     NewSocket.m_socket       = c;
     NewSocket.m_bConnected   = true;
     NewSocket.m_bCreated     = true;
}

void ServerSocket::Listen ( )
{
     Bind ( m_nPort );
     if (listen (m_socket, 5) == SOCKET_ERROR)
     {
          Except e ( _T("ServerSocket:: listen error\n") );
          throw e;
     }
}

bool ServerSocket::ClientName ( LPTSTR Host, int& nLen )
{
     return NameFromAddr ( &m_client_addr, Host, nLen);
}

bool ServerSocket::ClientIP ( LPTSTR HostIP, int& nLen )
{
     return IPFromAddr ( &m_client_addr, HostIP, nLen);
}

bool ServerSocket::ServerName ( LPTSTR Host, int& nLen )
{
     ResolveLocalName ( &m_resolved_name );
     return NameFromAddr ( &m_resolved_name, Host, nLen);
}

bool ServerSocket::ServerIP ( LPTSTR HostIP, int& nLen )
{
     ResolveLocalName ( &m_resolved_name );
     return IPFromAddr ( &m_resolved_name, HostIP, nLen);
```

```
}

bool ServerSocket::ResolveLocalName ( sockaddr_in * pAddr)
{
     TCHAR Name [STATIC_BUFFER];
     DWORD dwLen = STATIC_BUFFER;
     BOOL bSuccess;
     bSuccess = GetComputerName ( Name, &dwLen);
     hostent* pHost = NULL;
     ZeroMemory (pAddr, sizeof (sockaddr_in));
     if (false == bSuccess)
     {
          int cbLen = STATIC_BUFFER;
          bSuccess = (gethostname (Name, cbLen) !=
SOCKET_ERROR);
     }
     if (bSuccess)
     {
          pHost = gethostbyname (Name);
     }
     if (pHost)
     {
          CopyMemory (&pAddr->sin_addr, pHost->h_addr,
pHost->h_length);
          pAddr->sin_family = pHost->h_addrtype;
          return true;
     }
     else return false;
}
//////////////////////////////////////////////////////////
//////////////////
// Client Socket

void ClientSocket::Create ( )
{
     BaseSocket::Create ( );
}

void ClientSocket::Connect ( LPCTSTR Host, int nPort)
{
     if (false == m_bCreated)
          Create ( );
     ResolveName (nPort, Host);
     if (connect (m_socket, (sockaddr*)&m_addr,
sizeof(m_addr)) == SOCKET_ERROR)
     {
```

```
            return SurfaceDesc.lPitch;
      }
      return 0;
}

long WindowedVideo::GetBufferPitch ( )
{
      DD_CALL_INIT ( );
      if ( pScreen )
      {
            DDSURFACEDESC SurfaceDesc = {0};
            SurfaceDesc.dwSize = sizeof (SurfaceDesc);
            DD_CALL ( pOffscreen->GetSurfaceDesc (
&SurfaceDesc ) );
            return SurfaceDesc.lPitch;
      }
      return 0;
}

// Admin.cpp
#include <windows.h>
#include <ddraw.h>
#include <tchar.h>
#include <winsock.h>
#include "consultant.h"
#include "resource.h"
#include "socket.h"
#include "admin.h"
#include "cxport.h"
#include "crusher.h"
#include "gRect.h"
#include "diag.h"
#include "bitio.h"
#include "huff.h"
#include "ahuff.h"
#include "compress.h"
#include "hardware.h"
#include "adminvideo.h"
#include "comm.h"

BOOL CALLBACK MenuProc (HWND hDlg, UINT uMsg, WPARAM
wParam, LPARAM lParam);
HWND hMenu;

AdminConnection::AdminConnection ( )
{
      m_bConnected = false;
```

```
    m_hSignal = CreateEvent ( NULL, true, false,
ADMIN_EVENT);
    ResetEvent (m_hSignal);
    HDC hDC = CreateDC ( _T("DISPLAY"), NULL, NULL, NULL);
    m_HorzRes = GetDeviceCaps (hDC, HORZRES);
    m_VertRes = GetDeviceCaps (hDC, VERTRES);
    DeleteDC (hDC);

    m_refresh = false;

    m_hIconOn = LoadIcon (GetModuleHandle (NULL),
MAKEINTRESOURCE(IDI_ON)   );
    m_hIconOff     = LoadIcon (GetModuleHandle (NULL),
MAKEINTRESOURCE(IDI_OFF)  );
    m_hIconWait    = LoadIcon (GetModuleHandle (NULL),
MAKEINTRESOURCE(IDI_WAIT) );

}

AdminConnection::~AdminConnection ( )
{
    DestroyIcon (m_hIconOn);
    DestroyIcon (m_hIconOff);
    DestroyIcon (m_hIconWait);
}

void AdminConnection::InitClass ( )
{
    m_hWnd = NULL;
    m_HorzRes = 0;
    m_VertRes = 0;
    if (m_pBitmapInfo) delete m_pBitmapInfo;
    m_pBitmapInfo = NULL;

    // direct draw objects
    pDirectDraw = NULL;
    pSurface = NULL;
}


////////////////////////////////////////////////////////////
////////////////////
//


int AdminConnection::HorzRes ( )
{
```

```
     return m_HorzRes;
}

int AdminConnection::VertRes ( )
{
     return m_VertRes;
}

int AdminConnection::Connect ( LPCTSTR Host )
{
     TRACE("Attempting to connect.\n");
     int bResult = CONNECT_NOT_AVAILABLE;
     try
     {
          HCURSOR hCursor  = LoadCursor (NULL, IDC_WAIT);
          HCURSOR hCurrent = SetCursor (hCursor);

          SendMessage (GetDlgItem (m_hDlg, IDC_CONNECTION),
STM_SETIMAGE, ICON_BIG,
               (WPARAM)m_hIconWait);
          ResetEvent (m_hSignal);
          m_VideoSocket.Connect (Host, VIDEO_PORT);
          m_InputSocket.Connect (Host, INPUT_PORT);
          TRACE ("Connected.\n");
          m_bConnected = true;
          if (ConnectionAccepted ( ) == false)
          {
               Disconnect ( );
               SendMessage (GetDlgItem (m_hDlg,
IDC_CONNECTION), STM_SETIMAGE, ICON_BIG,
                    (WPARAM)m_hIconOff);
               return CONNECT_NOT_AVAILABLE;
          }
          TRACE ("Agent available.\n");
          int v_handshake = VideoHandshake ( );
          int i_handshake = InputHandshake ( );
          if (v_handshake == false ||
               i_handshake == false)
          {
               SendMessage (GetDlgItem (m_hDlg,
IDC_CONNECTION), STM_SETIMAGE, ICON_BIG,
                    (WPARAM)m_hIconOff);
               return CONNECT_INCORRECT_VERSION;
          }
          if (v_handshake == -1 || i_handshake == -1)
          {
```

```
                SendMessage (GetDlgItem (m_hDlg,
IDC_CONNECTION), STM_SETIMAGE, ICON_BIG,
                     (WPARAM)m_hIconOff);
                return CONNECT_AGENT_REJECT;
           }
           SetCursor  (hCurrent);

           SendMessage (GetDlgItem (m_hDlg, IDC_CONNECTION),
STM_SETIMAGE, ICON_BIG,
                (WPARAM)m_hIconOn);

           m_hWnd = CreateWindow (// WS_EX_TOPMOST,
                MAIN_WND_CLASS, _T("Open emulation session
for eSC"),
                WS_VISIBLE | WS_POPUPWINDOW,
                0, 0, m_HorzRes, m_VertRes,
                NULL, NULL, GetModuleHandle (NULL), NULL);

           //hMenu = CreateDialog ( GetModuleHandle ( NULL
), MAKEINTRESOURCE(IDD_MENU),
           //   m_hWnd, (DLGPROC)MenuProc );
           //SetWindowPos ( hMenu, HWND_TOPMOST, 0, 0, 0, 0,
SWP_NOMOVE | SWP_NOSIZE );
           //SetClassLong ( hMenu, GCL_STYLE, GetClassLong (
hMenu, GCL_STYLE )
           //   | CS_HREDRAW | CS_SAVEBITS | CS_VREDRAW );

           // hide the main window, wait on the threads,
show the window again
           DWORD dwThreadID;
           DWORD dwVideoExitCode;
           m_Thread[0] = CreateThread ( NULL, 0,
(LPTHREAD_START_ROUTINE)VideoLoopProxy,
                (LPVOID)this, 0, &dwThreadID);

           // ShowWindow (m_hDlg, SW_HIDE);

           InputLoop ( );
           if (WaitForSingleObject (m_Thread[0], 5000) ==
WAIT_TIMEOUT)
           {
                TRACE ("Video Thread did not exit, killing
it now\n");
                TerminateThread (m_Thread [0],
VIDEO_EXIT_HANG );
                Disconnect ( );
           }
```

```
          GetExitCodeThread ( m_Thread[0], &dwVideoExitCode
);
          // ShowWindow (m_hDlg, SW_SHOW);

          SendMessage (GetDlgItem (m_hDlg, IDC_CONNECTION),
STM_SETIMAGE, ICON_BIG,
               (WPARAM)m_hIconOff);

          if (VIDEO_EXIT_HARDWARE_ERROR      ==
dwVideoExitCode ||
               VIDEO_EXIT_DIRECT_DRAW_ERROR  ==
dwVideoExitCode)
               return CONNECT_HARDWARE_INCOMPATIBLE;
          if (VIDEO_EXIT_CLIENT_DOESNT_SUPPORT ==
dwVideoExitCode)
               return CONNECT_CLIENT_INCOMPATIBLE;
          if (VIDEO_EXIT_HANG                  ==
dwVideoExitCode)
               return CONNECT_VIDEO_HANG;
          bResult = CONNECT_SUCCESS;
     }
     catch (Except e)
     {
          e.Trace ( );
          Disconnect ( );
     }
     return bResult;
}

BOOL CALLBACK MenuProc (HWND hDlg, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
     switch (uMsg)
     {
     case WM_INITDIALOG:
          return true;
     case WM_COMMAND:
          {
               WORD wNotifyCode = HIWORD(wParam);
               WORD wID = LOWORD(wParam);
               HWND hwndCtl = (HWND) lParam;
               switch (wID)
               {
               case ID_END_SESSION:
                    {
                    /*
```

```
                    HANDLE hSignal = OpenEvent
(EVENT_ALL_ACCESS, false, ADMIN_EVENT);
                    SetEvent (hSignal);
                    CloseHandle (hSignal);
                    */
                    PostMessage ( GetParent ( hDlg ),
WM_KEYDOWN, VK_F12, 0 );
                    }
                    break;
               case ID_MINIMIZE:
                    ShowWindow (GetParent ( hDlg ),
SW_MINIMIZE);
                    break;
               }
          }
          return true;
     }
     return false;
}

void AdminConnection::Disconnect (LPCTSTR pMsg/*=NULL*/,
bool bDisplayMsg/*=false*/ )
{
     TRACE ("Disconnect called.\n");
     HANDLE hSignal = OpenEvent (EVENT_ALL_ACCESS, false,
ADMIN_EVENT);
     SetEvent (hSignal);
     CloseHandle (hSignal);

     // network state
     m_VideoSocket.Close ( );
     m_InputSocket.Close ( );
}

bool AdminConnection::CreateControlDialog ( )
{
     m_hDlg = CreateDialog (GetModuleHandle (NULL),
MAKEINTRESOURCE (IDD_ADMIN_EMULATION),
                  NULL, (FARPROC)AdminDlgProc);
     // SetWindowText (GetDlgItem (m_hDlg, IDC_BUILD),
__TIMESTAMP__);
     return true;
}


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

```
//
// Protocol for mouse and keyboard: send over the command,
then the structure
// populated with information about the command

////////////////////////////////////////////////////////////
////////////////////////////
//  Mouse Control

void AdminConnection::MouseButton ( UINT uButton )
{
     // tell the other side the event to occur
     MouseEvent me = {0};
     DWORD com = INPUT_MOUSE;
     // check for double clicks
     if (uButton == WM_LBUTTONDBLCLK   || uButton ==
WM_RBUTTONDBLCLK   ||
          uButton == WM_MBUTTONDBLCLK   || uButton ==
WM_NCLBUTTONDBLCLK ||
          uButton == WM_NCRBUTTONDBLCLK || uButton ==
WM_NCMBUTTONDBLCLK)
          com = INPUT_DOUBLE_CLICK_MOUSE;
     InputSend ((LPBYTE)&com, sizeof(DWORD));
     switch (uButton)
     {
     case WM_LBUTTONDOWN:      case WM_NCLBUTTONDOWN:
     case WM_LBUTTONDBLCLK:    case WM_NCLBUTTONDBLCLK:
          me.dwFlags = MOUSEEVENTF_LEFTDOWN;
          InputSend ((LPBYTE)&me, sizeof (me));   break;
     case WM_RBUTTONDOWN:      case WM_NCRBUTTONDOWN:
     case WM_RBUTTONDBLCLK:    case WM_NCRBUTTONDBLCLK:
          me.dwFlags = MOUSEEVENTF_RIGHTDOWN;
          InputSend ((LPBYTE)&me, sizeof (me));   break;
     case WM_MBUTTONDOWN:      case WM_NCMBUTTONDOWN:
     case WM_MBUTTONDBLCLK:    case WM_NCMBUTTONDBLCLK:
          me.dwFlags = MOUSEEVENTF_MIDDLEDOWN;
          InputSend ((LPBYTE)&me, sizeof (me));   break;
     default:       break;
     }
     // button UP
     switch (uButton)
     {
     case WM_LBUTTONUP:        case WM_NCLBUTTONUP:
     case WM_LBUTTONDBLCLK:    case WM_NCLBUTTONDBLCLK:
          me.dwFlags = MOUSEEVENTF_LEFTUP;
          InputSend ((LPBYTE)&me, sizeof (me));   break;
     case WM_RBUTTONUP:        case WM_NCRBUTTONUP:
```

```
        case WM_RBUTTONDBLCLK:    case WM_NCRBUTTONDBLCLK:
             me.dwFlags = MOUSEEVENTF_RIGHTUP;
             InputSend ((LPBYTE)&me, sizeof (me));    break;
        case WM_MBUTTONUP:        case WM_NCMBUTTONUP:
        case WM_MBUTTONDBLCLK:    case WM_NCMBUTTONDBLCLK:
             me.dwFlags = MOUSEEVENTF_MIDDLEUP;
             InputSend ((LPBYTE)&me, sizeof (me));    break;
        default:        break;
        }
}

// Mouse coordinates must be sent over in the range from 0
- 65k
// Using the constant 0xffff for 65k

void AdminConnection::MouseMove ( UINT uMsg, LPARAM lParam
)
{
        MouseEvent me = {0};
        DWORD com = INPUT_MOUSE;
        InputSend ((LPBYTE)&com, sizeof(DWORD));
        if (WM_MOUSEMOVE == uMsg)
        {
             POINT pt = {LOWORD (lParam), HIWORD (lParam)};
             ClientToScreen (m_hWnd, &pt);
             me.dx = (DWORD)((float ((float)pt.x /
(float)HorzRes ( )))
                  * (float)MOUSE_X);
             me.dy = (DWORD)((float ((float)pt.y /
(float)VertRes ( )))
                  * (float)MOUSE_Y);
        }
        else
        {
             POINTS pt = MAKEPOINTS (lParam);
             me.dx = (DWORD)((float ((float)pt.x /
(float)HorzRes ( )))
                  * (float)MOUSE_X);
             me.dy = (DWORD)((float ((float)pt.y /
(float)VertRes ( )))
                  * (float)MOUSE_Y);
        }
        me.dwFlags = MOUSEEVENTF_ABSOLUTE | MOUSEEVENTF_MOVE;
        InputSend ((LPBYTE)&me, sizeof (me));
}

// tell the client to send over the whole screen
```

```
void AdminConnection::CommandMsg ( int nEvent )
{
     DWORD com = 0;
     switch (nEvent)
     {
     case USER_PAUSE:
          com = INPUT_PAINTING_PAUSE;
          break;
     case USER_RESUME:
          com = INPUT_PAINTING_RESUME;
          break;
     }
     if (com != 0)
          InputSend ((LPBYTE)&com, sizeof(DWORD));
}


////////////////////////////////////////////////////////////
////////////
// Keyboard

void AdminConnection::Keystroke ( UINT uMsg, WPARAM VkCode
)
{
     // handle hotkeys //
     if (VkCode == VK_F12 || VkCode == VK_F9)
     {
          SetEvent (m_hSignal);
          return;
     }
     if (VkCode == VK_F11)
     {
          ShowWindow (m_hWnd, SW_MINIMIZE);
     }
     if (VkCode == VK_F10)
     {

     }
     KeyboardEvent ke = {0};
     DWORD com = INPUT_KEYBOARD;
     InputSend ((LPBYTE)&com, sizeof(DWORD));
     if (WM_KEYUP == uMsg || WM_SYSKEYUP == uMsg)
          ke.dwFlags = KEYEVENTF_KEYUP;
     ke.Vk = VkCode;
     InputSend ((LPBYTE)&ke, sizeof (ke));
}
```

```
void AdminConnection::HotKey ( int nId )
{
     OtherEvent se = {0};
     DWORD com = INPUT_HOTKEY;
     InputSend ((LPBYTE)&com, sizeof(DWORD));
     se.HotKeyId = nId;
     InputSend ((LPBYTE)&se, sizeof (se));
}

void AdminConnection::RegisterHotKeys ( )
{
     BOOL_CALL (RegisterHotKey (m_hWnd, HOTKEY_ALTTAB,
MOD_ALT, VK_TAB));
     BOOL_CALL (RegisterHotKey (m_hWnd, HOTKEY_CTRLALTDEL,
MOD_ALT | MOD_CONTROL, VK_DELETE));
     BOOL_CALL (RegisterHotKey (m_hWnd, HOTKEY_CTRLESC,
MOD_CONTROL, VK_ESCAPE));
}

void AdminConnection::ClearHotKeys ( )
{
     UnregisterHotKey (m_hWnd, HOTKEY_CTRLESC);
     UnregisterHotKey (m_hWnd, HOTKEY_CTRLALTDEL);
     UnregisterHotKey (m_hWnd, HOTKEY_ALTTAB);

}
////////////////////////////////////////////////////////////
//////////////////////////
// Video Connection Information


bool AdminConnection::VerifyDirectDraw ( )
{
     return true;
}

int AdminConnection::VideoLoopProxy ( LPVOID pThis )
{
     return ((AdminConnection*)pThis)->GridVideoLoop ( );
}


////////////////////////////////////////////////////////////
////////////////////

int AdminConnection::GridVideoLoop ( )
```

```
{
     int bm_len = 0;

     TRACE ("Entering (grid) Video Loop\n");
     HANDLE hSignal = OpenEvent (SYNCHRONIZE, false,
ADMIN_EVENT);

     // first thing sent over is a BITMAPINFO structure
     HardwareInfo info;
     VideoRecv ((LPBYTE)&info, sizeof (info));
     m_VertRes = info.ScreenHeight;
     m_HorzRes = info.ScreenWidth;
     if ( info.ByteCount != 1 )
     {
          TRACE ( "not in 8bpp mode\n" );
     }

     if (info.GetFail ( ))
     {
          Disconnect ( );
          return VIDEO_EXIT_CLIENT_DOESNT_SUPPORT;
     }

     LPPALETTEENTRY pPal = NULL;
     if ( info.ByteCount == 1 )
     {
          // get the palette
          pPal = new PALETTEENTRY [256];
          VideoRecv ( (LPBYTE)pPal,   256 * sizeof
(PALETTEENTRY));
     }

     AdminVideo video;

     if (video.OpenSession ( info, m_hWnd, pPal) == false)
     {
          TRACE ("Open session failed\n");
          Disconnect ( );
          return VIDEO_EXIT_DIRECT_DRAW_ERROR;
     }

     DirtyBlock * arrayDirty = new DirtyBlock
[info.MaxGridCount];
     LPBYTE pCompressedBuffer = new
BYTE[video.TotalBufferSize ( )];

     InfoBlock header;
```

```
     Status status;

     bool bLost = false;
     bool bContinue = true;
     while (bContinue)
     {
          if (WaitForSingleObject (hSignal, 0) ==
WAIT_OBJECT_0)
          {
               TRACE("Event signaled - Video Loop\n");
               break;
          }
          // send over status information
          status.Clear ( );
          if (video.Pause ( ))     status.SetPause ( );
          if (video.Refresh ( ) || m_refresh)     {
               m_refresh = false;
               TRACE ("Sending a refresh message\n");
               status.SetRefresh ( );}
          VideoSend ( (LPBYTE)&status, sizeof (status));

          VideoRecv ( (LPBYTE)&header,   sizeof (header));

          if ( InfoBlock::PALETTE_AVAIL & header.fCommands)
          {
               VideoRecv ( (LPBYTE)pPal,   256 * sizeof
(PALETTEENTRY));
               video.SetPalette ( pPal );
          }
          if (VIDEO_NO_PAINT == header.fStatus)
     continue;
          if (VIDEO_CLOSE_CONNECTION == header.fStatus)
          {
               bContinue = false;
               continue;
          }
          VideoRecv ( (LPBYTE)arrayDirty, sizeof
(DirtyBlock) * header.nDirtyCount);
          VideoRecv ( pCompressedBuffer,
header.cbCompressedSize);

          video.ProcessFrame ( header, arrayDirty,
pCompressedBuffer, 0 );

          RedrawWindow ( hMenu, NULL, NULL, RDW_ERASE |
RDW_FRAME | RDW_INVALIDATE );
     }
```

```
     // clean up
     delete []pCompressedBuffer;
    delete []m_pBitmapInfo;
     delete []pPal;

     CloseHandle (hSignal);
     Disconnect ( );
     TRACE ("VideoLoop exiting\n");
    return 1;
}

////////////////////////////////////////////////////////////
//////////////////
// Input loop

int AdminConnection::InputLoopProxy ( LPVOID pThis )
{
     return ((AdminConnection*)pThis)->InputLoop ( );
}

int AdminConnection::InputLoop ( )
{
     TRACE ("Entering Input Loop\n");
     HANDLE hSignal = OpenEvent (SYNCHRONIZE, false,
ADMIN_EVENT);
     RegisterHotKeys ( );
     MSG move_message, msg;
     bool bFlushMoves   = false;
     BOOL bKeepChecking = true;
     int nCount = 0;
     bool bQuit = false;
     while ( false == bQuit )
     {
          // wait for a message or the signal of the event
          //if (false == bQuit && WaitForSingleObject (
hSignal, 0 ) == WAIT_OBJECT_0)
          if (MsgWaitForMultipleObjects (1, &hSignal,
false, 3000, QS_ALLINPUT) == WAIT_OBJECT_0)
          {
               DWORD com = INPUT_CLOSE_CONNECTION;
               InputSend ((LPBYTE)&com, sizeof(DWORD));
               TRACE ("Event signaled - input loop\n");
               TRACE ("Destroying Window\n");
               ClearHotKeys ( );
               DestroyWindow (m_hWnd);
               bQuit = true;
          }
```

```
          //while (HIWORD (GetQueueStatus ( QS_ALLINPUT ))
& QS_ALLINPUT )
          bKeepChecking = true;
          while (bKeepChecking)
          {
               while (PeekMessage (&msg, NULL, 0, 0,
PM_REMOVE))
               {
                    TranslateMessage (&msg);
                    // the goal with this convoluted loop
is to bunch moves
                    // together into one message rather
then sending each one
                    if (WM_MOUSEMOVE == msg.message)
                    {
                         move_message = msg;
                         bFlushMoves = true;
                         Sleep (200);
                    }
                    else
                    {
                         if (bFlushMoves)
                         {
                              bFlushMoves = false;
                              DispatchMessage
(&move_message);
                         }
                         DispatchMessage  (&msg);
                    }
               }
               bKeepChecking = GetInputState( );
          }
          // if we still have a cahed move, send it now
          if (bFlushMoves)
          {
               bFlushMoves = false;
               DispatchMessage (&move_message);
          }
     }
     CloseHandle (hSignal);
     TRACE ("InputLoop exiting\n");
     return 0;
}
```

```
//////////////////////////////////////////////////////////
//
// Network IO

bool AdminConnection::ConnectionAccepted ( )
{
     if (m_VideoSocket.CanRead ( ) && m_InputSocket.CanRead
( )) return true;
     else return false;
}

int AdminConnection::VideoHandshake ( )
{
     int AgentVersion = 0, AdminVersion = VERSION;
     VideoRecv ( (LPBYTE)&AgentVersion, sizeof (int) );
     VideoSend ( (LPBYTE)&AdminVersion, sizeof (int) );
     if (AgentVersion == REJECT)
     {
          return -1;
     }
     if (AgentVersion != AdminVersion)
     {
          TRACE ("Video version wrong.\n");
          return false;
     }
     TRACE ("Video Handshake Success.\n");
     return true;
}

int AdminConnection::InputHandshake ( )
{
     int AgentVersion = 0, AdminVersion = VERSION;
     InputRecv ( (LPBYTE)&AgentVersion, sizeof (int) );
     InputSend ( (LPBYTE)&AdminVersion, sizeof (int) );
     if (AgentVersion == REJECT)
     {
          return -1;
     }
     if (AgentVersion != AdminVersion)
     {
          TRACE ("Input version wrong.\n");
          return false;
     }
     TRACE ("Input Handshake Success.\n");
     return true;
}
```

```
////////////////////////////////////////////////////////////
//
// Sending and Receiving

void AdminConnection::InputSend (LPBYTE pMsg, int len)
{
     if (m_bConnected)
     {
          int nRes = 0;
          try
          {
               nRes = m_InputSocket.SendFully (pMsg, len);
          }
          catch (Except e)
          {
               TRACE ("Input Socket Send Failed\n");
               e.Trace ( );
               Disconnect ( );
          }
          if (0 == nRes) Disconnect ( );
     }
}

void AdminConnection::InputRecv (LPBYTE pMsg, int len)
{
     if (m_bConnected)
     {
          int nRes = 0;
          try
          {
               nRes = m_InputSocket.RecvFully (pMsg, len);
          }
          catch (Except e)
          {
               TRACE ("Input Socket Recv Failed\n");
               e.Trace ( );
               Disconnect ( );
          }
          if (0 == nRes) Disconnect ( );
     }
}

void AdminConnection::VideoSend (LPBYTE pMsg, int len)
{
     if (m_bConnected)
```

```
    {
        int nRes = 0;
        try
        {
            nRes = m_VideoSocket.SendFully (pMsg, len);
        }
        catch (Except e)
        {
            TRACE ("Video Socket Send Failed\n");
            e.Trace ( );
            Disconnect ( );
        }
        if (0 == nRes) Disconnect ( );
    }
}

void AdminConnection::VideoRecv (LPBYTE pMsg, int len)
{
    if (m_bConnected)
    {
        int nRes = 0;
        try
        {
            nRes = m_VideoSocket.RecvFully (pMsg, len);
        }
        catch (Except e)
        {
            TRACE ("Video Socket Recv Failed\n");
            e.Trace ( );
            Disconnect ( );
        }
        if (0 == nRes) Disconnect ( );
    }
}

////////////////////////////////////////////////////////////////
///////

#include <windows.h>
#include <tchar.h>
#include <ddraw.h>
#include "consultant.h"
#include "crusher.h"
#include "gRect.h"
#include "rle.h"
#include "diag.h"
#include "bitio.h"
```

```
#include "huff.h"
#include "ahuff.h"
#include "compress.h"
#include "hardware.h"
#include "windowed_hardware.h"
#include "checksum.h"
#include "adminvideo.h"


AdminVideo::AdminVideo ( )
{
     m_ByteCount = 1;
}

AdminVideo::~AdminVideo ( )
{
     CloseSession ( );
}

bool AdminVideo::OpenSession ( const HardwareInfo& info,
HWND hWnd, LPPALETTEENTRY lpPalette)
{
     ProcessInfo ( info );
     m_hWnd = hWnd;
     // open the direct draw objects
     if ( m_display.Open ( m_ScreenWidth, m_ScreenHeight,
                                m_OffscreenWidth,
m_OffscreenHeight,
                                Video::SCREEN_ADMIN,
m_ByteCount, m_hWnd, lpPalette) == false)
          return false;
     // set up the parameters for the work to be done
     m_rctScreen         = Rect  ( m_ScreenWidth,
m_ScreenHeight, GRID_WIDTH, GRID_HEIGHT);
     m_rctOffscreen = Rect  ( (m_OffscreenWidth -
m_padding), m_OffscreenHeight,
          OFFSCREEN_WIDTH, (GRID_COUNT / OFFSCREEN_WIDTH));
     m_cbRowBufferSize    = m_OffscreenWidth *
(m_ScreenHeight / GRID_HEIGHT) * m_ByteCount;
     m_cbTotalBufferSize  = m_cbRowBufferSize * (GRID_COUNT
/ OFFSCREEN_WIDTH);
     return true;
}



void AdminVideo::CloseSession ( )
```

```
{
     m_display.Close ( );
}


void AdminVideo::ProcessInfo ( const HardwareInfo& info )
{
     m_ScreenWidth       = info.ScreenWidth;
     m_ScreenHeight      = info.ScreenHeight;

     m_padding = PADDING * (m_ScreenWidth /
PADDING_DIVISOR);
     m_OffscreenWidth  = ((m_ScreenWidth  / GRID_WIDTH) *
OFFSCREEN_WIDTH) + m_padding;
     m_OffscreenHeight = ( m_ScreenHeight / GRID_HEIGHT) *
(GRID_COUNT / OFFSCREEN_WIDTH);
     m_ByteCount = info.ByteCount;
}

bool AdminVideo::ProcessFrame ( InfoBlock& header,
DirtyBlock* arrayDirty,
                                        LPBYTE pComp, DWORD
fCommands)
{
     if (m_bLost)   RestoreLostSurface ( );
     bool bResult = false;
     if (ExpandBuffer ( header, pComp ) == true )
          bResult = ProcessIteration ( header, arrayDirty,
fCommands );
     return bResult;
}


bool AdminVideo::ProcessIteration ( InfoBlock& header,
DirtyBlock* arrayDirty, DWORD fCommands )
{
     m_rctOffscreen.MoveFirst ( );
     for (int nIndex = 0; nIndex < header.nDirtyCount;
nIndex ++)
     {
          m_rctScreen.MoveTo (arrayDirty[nIndex].xPos,
arrayDirty[nIndex].yPos);
          if (m_display.PutScreenRect ( m_rctScreen,
m_rctOffscreen ) == false)
               {
                    m_bLost = true;
                    return false;
```

```
            }
        m_rctOffscreen.MoveNext ( );
    }
    return true;
}


bool AdminVideo::ExpandBuffer ( InfoBlock& header, LPBYTE
pOut )
{
    // get the video buffer and compress
    LPBYTE pOffscreen;
    if (m_display.GetBufferMemory (
m_rctOffscreen.FullArea ( ), pOffscreen ) == false )
    {
        TRACE ("Unable to get buffer memory\n");
        return false;
    }
    if (m_compressionEngine.Expand ( pOut,
        pOffscreen, header.cbFullSize,
header.cbCompressedSize,
        header.fCompression) == false)
    {
        TRACE ("Compression failed\n");
        return    false;
    }
    return true;
}

bool AdminVideo::SetPalette ( LPPALETTEENTRY pPal )
{
    return m_display.SetEntries ( pPal, 256 );
}

bool AdminVideo::RestoreLostSurface ( )
{
    bool bResult;
    bResult = m_display.RestoreLostSurface ( );
    if ( bResult )
    {
        m_bRefresh = true;
        m_bLost = false;
    }
    return bResult;
}

// emulationAdmin.cpp
```

```
                    "Smart Consultant Administrator
v4.0\r\n"
                    "\r\nNo host specified.\r\n"
                    "\r\nUsage:\r\n\t/T <TargetName>", "e-
Parcel", MB_OK );
          return 0;
     }
     OpenLogFile ("Admin");
     TRACE ("Entering WinMain for the Admin\n");
     InitializeInstance ( );
     Admin.CreateControlDialog ( );
     MSG msg;

     while (GetMessage (&msg, NULL, 0, 0))
     {
          if ( !IsDialogMessage (Admin.m_hDlg, &msg) )
          {
               TranslateMessage (&msg);
               DispatchMessage (&msg);
          }
     }
     return 0;
}

void ThreadFunc ( HWND );

BOOL CALLBACK AdminDlgProc (HWND hDlg, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
     switch (uMsg)
     {
     case WM_INITDIALOG:
          PostMessage ( hDlg, USER_CONNECT, 0, 0 );
          return true;
     case USER_CONNECT:
          {
          DWORD id;
          CreateThread ( NULL, 0,
(LPTHREAD_START_ROUTINE)ThreadFunc, (LPVOID)hDlg, 0, &id );
          }
          return true;
     case WM_COMMAND:
          DlgCommand ( hDlg, wParam, lParam);
          return true;
     case WM_DESTROY:
          TRACE ( "Emulation leaving\n" );
          PostQuitMessage ( 0 );
```

```
            return true;
      default:
            return false;
      }
}

void ThreadFunc ( HWND hDlg )
{
      // grab the name of the client from the command line
      TRACE ( host );

      SetWindowText ( GetDlgItem ( hDlg, IDC_CLIENT_NAME ),
host );
      SetWindowText ( GetDlgItem ( hDlg, IDC_STATUS),
_T("Connecting...") );
      int nResult = Admin.Connect ( host );
      SetWindowText ( GetDlgItem ( hDlg, IDC_STATUS),
_T("Disconnecting...") );

      HICON hIcon = LoadIcon ( NULL, IDI_WARNING );
      switch (nResult)
            {
            case CONNECT_SUCCESS:
                  SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                        _T("Connection closed.") );
                  PostQuitMessage ( 0 );
                  break;
            case CONNECT_NOT_AVAILABLE:
                  SendMessage (GetDlgItem (hDlg,
IDC_STATUS_ICON), STM_SETIMAGE, ICON_BIG,
                        (LPARAM)hIcon);
                  SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                        _T("Unable to connect.\r\nThe client is
either busy or not running.") );
                  return;
            case CONNECT_INCORRECT_VERSION:
                  SendMessage (GetDlgItem (hDlg,
IDC_STATUS_ICON), STM_SETIMAGE, ICON_BIG,
                        (LPARAM)hIcon);
                  SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                        _T("Unable to connect.\r\nClient's
version does not match.") );
                  return;
            case CONNECT_AGENT_REJECT:
```

```
                SendMessage (GetDlgItem (hDlg,
IDC_STATUS_ICON), STM_SETIMAGE, ICON_BIG,
                     (LPARAM)hIcon);
                SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                     _T("Your connection has been rejected
by the client.") );
                return;
           case CONNECT_HARDWARE_INCOMPATIBLE:
                SendMessage (GetDlgItem (hDlg,
IDC_STATUS_ICON), STM_SETIMAGE, ICON_BIG,
                     (LPARAM)hIcon);
                SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                     _T("You are unable to emulate the
client's hardware.\r\nThe most likely cause "
                        "is that the client has a higher
screen resolution.") );
                return;
           case CONNECT_CLIENT_INCOMPATIBLE:
                SendMessage (GetDlgItem (hDlg,
IDC_STATUS_ICON), STM_SETIMAGE, ICON_BIG,
                     (LPARAM)hIcon);
                SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                     _T("The client does not support the
neccessary video mode.\r\nThe most likely "
                        "cause is that the client is not in
256 color mode.") );
                return;
           case CONNECT_VIDEO_HANG:
                SetWindowText ( GetDlgItem ( hDlg,
IDC_STATUS ),
                     _T("Video connection was unable to
disconnect properly.") );
                return;
           }
     Sleep ( 3000 );
     PostMessage ( hDlg, WM_DESTROY, 0, 0 );
}

void DlgCommand (HWND hDlg, WPARAM wParam, LPARAM lParam)
{
     WORD wNotifyCode = HIWORD(wParam);
     WORD wID = LOWORD(wParam);
     HWND hwndCtl = (HWND) lParam;
     switch (wID)
```

```
      {
      case ID_CLOSE:
            EndDialog ( hDlg, 0 );
            PostQuitMessage ( 0 );
            break;
      }
}



void InitializeInstance ( )
{
      // start up the winsock stuff
      WSADATA ws;
      WSAStartup (0x0101, &ws);

      // Create and Register window class
      WNDCLASS wc = {0};
      wc.style = CS_DBLCLKS;
      wc.hInstance = GetModuleHandle (NULL);
      wc.hbrBackground = GetStockObject (BLACK_BRUSH);
      wc.lpszClassName = MAIN_WND_CLASS;
      wc.lpfnWndProc   = WndProc;
      wc.hCursor           = LoadCursor (NULL, IDC_ARROW);
      RegisterClass (&wc);
}



////////////////////////////////////////////////////////////
///////////////
// Wnd Proc for blank screen window, on which the other
machine is projected


LRESULT CALLBACK WndProc (HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
      switch (uMsg)
      {
      // disallow the user to move the window

      case WM_MOVING:
            {
                  LPRECT pRect = (LPRECT) lParam;
                  pRect->left          = 0; pRect->top
      = 0;
```

```
                pRect->bottom  = Admin.VertRes ( );
                pRect->right   = Admin.HorzRes ( );
                return true;
            }
            break;

        ////////////////////////////////////////////////////////
////////////////////
        // Mouse Messages that are passed to the other side
        case WM_NCMOUSEMOVE:
        case WM_MOUSEMOVE:
            Admin.MouseMove (uMsg, lParam);
            break;
        case WM_NCLBUTTONDOWN:
        case WM_LBUTTONDOWN:
        case WM_NCLBUTTONUP:
        case WM_LBUTTONUP:
        case WM_NCRBUTTONDOWN:
        case WM_RBUTTONDOWN:
        case WM_NCRBUTTONUP:
        case WM_RBUTTONUP:
        case WM_LBUTTONDBLCLK:
        case WM_RBUTTONDBLCLK:
            Admin.MouseButton (uMsg);
            break;
        ////////////////////////////////////////////////////////
////////////////////
        // Keyboard messages that are passed to the other side
        case WM_KEYDOWN:
        case WM_KEYUP:
        case WM_SYSKEYDOWN:
        case WM_SYSKEYUP:
            Admin.Keystroke (uMsg, wParam);
            break;
        case WM_HOTKEY:
            TRACE ("HotKey\n");
            ShowWindow (hWnd, SW_SHOWMAXIMIZED);
            Admin.HotKey ( wParam );
            break;
        case WM_DESTROY:
            return true;
        case USER_RESUME:
        case USER_PAUSE:
            Admin.CommandMsg ( (int)uMsg );
            break;
        case WM_SETFOCUS:
            Admin.SetRefresh ( );
```

```
            break;
      ////////////////////////////////////////////////////////
//////////////////////
      default:
            return DefWindowProc (hWnd, uMsg, wParam,
lParam);
      }
      return 0;
}

LRESULT CALLBACK FullScreenWndProc (HWND hWnd, UINT uMsg,
WPARAM wParam, LPARAM lParam)
{
      /*
      switch (uMsg)
      {
      default:
            return DefWindowProc (hWnd, uMsg, wParam,
lParam);
      }
      */
      return DefWindowProc (hWnd, uMsg, wParam, lParam);
}
```

Having above indicated several embodiments of the Subject Invention, it will occur to those skilled in the art that modifications and alternatives can be practiced within the spirit of the invention. It is accordingly intended to define the scope of the invention only as indicated in the following claims.

WHAT IS CLAIMED IS:

1. A system for minimizing the overall screen refresh time of a screen at an administrator's computer in which the computer at the administrator's side remotely controls a computer at a user's side, with said computers being interconnected over a network, comprising:

means at said user's side for selecting one of a number of compression algorithms for compressing data at said user's side prior to transmission over said network to said administrator's side, said selecting means, including means for determining the compression time and the transmission time of said data, means for calculating the ratio of said compression time to said transmission time and means for selecting that compression algorithm which results in said ratio approaching one, thus to reduce screen refresh time at said administrator's computer.

2. The system of Claim 1, wherein said compression algorithms are ordered with the highest compression rate first, and wherein said selection means first picks that algorithm which has the highest compression rate.

3. The system of Claim 2, wherein said selecting means monitors said ratio and proceeds to an algorithm having the next lower compression rate until said ratio approaches one.

4. The system of Claim 3, wherein said screen has a refresh cycle and wherein said selecting means monitors said ratio for every refresh cycle.

5. The system of Claim 1, wherein said screen is divided into grids and further including means for generating a check sum for each grid, means for comparing a grid check sum to the previous check sum, means for detecting a change in check sum for a grid, means responsive to a change in check sum for collecting and compressing the corresponding data, and means for checking the entire screen on a grid by grid basis to complete a screen refresh cycle.

6. The system of Claim 1, and further including means for changing said transmission rate as well as said compression rate to drive said ratio to one.

7. The system of Claim 1, wherein said algorithms include run length encoding algorithms.

8. The system of Claim 1, wherein the lowest compression rate algorithm includes a run length encoding algorithm.

9. The system of Claim 8, wherein the algorithm having the next higher compression rate, includes a Huffman compression algorithm preceded by run length encoding.

10. The system of Claim 9, wherein the algorithm having the next higher compression rate includes a modified adaptive Huffman compression algorithm using a 9-bit tree entry size, with said Huffman algorithm preceded by run length encoding.

11. The system of Claim 10, wherein said tree entry size is a 13-bit tree entry size corresponding to the highest compression rate.

12. A system for minimizing the screen refresh time of a display in communication with a first computer over a network, said system comprising:

a compression algorithm selector on said first computer selecting one of a plurality of a compression algorithms for compressing data at said first computer prior to transmission of said data over said network to said display, said compression algorithm selector comprising:

a calculator determining the compression time and the transmission time of said data and calculating the ratio of said compression time to said transmission time for said selected one of said plurality of compression algorithms; and

a selector selecting that compression algorithm of said pluality of of compression algorithms which results in said ratio approaching a predetermined value. --

13. The system of Claim 12, wherein said plurality of compression algorithms are ordered according to compression rate, and wherein said compression algorithm selector first picks that compression algorithm of said plurality of compression algorithms having the highest compression rate.

14. The system of Claim 13, wherein compression algorithm selector iteratively selects said said compression algorithm having the next lower compression rate until said ratio approaches said predetermined value.

15. The system of Claim 12, wherein said display is divided into a plurality of grids and wherein said compression algorithm selector further comprises:

a check sum generator generating a check sum for each grid of said plurality of grids;

a comparator comparing the check sum of each grid to the previous check sum for said grid of said plurality of grids and detecting a change in check sum for each grid of said plurality of grids; and

a data compressor collecting and compressing data for each grid having a change in checksum, for each grid of said plurality of grids.

16. The system of Claim 12, further comprising a variable transmitter capable of changing said transmission rate.

17. A method for minimizing the screen refresh time of a display in communication with a first computer over a network, said method comprising the steps of:

selecting one of a plurality of a compression algorithms for compressing data at said first computer prior to transmission of said data over said network to said display;

determining the compression time and the transmission time of said data;

calculating the ratio of said compression time to said transmission time for said selected one of said plurality of compression algorithms; and

selecting that compression algorithm of said pluality of of compression algorithms which results in said ratio approaching [one] a predetermined value.

18. The method of Claim 17 further comprising the steps of:

ordering said plurality of compression algorithms are ordered according to compression rate; and

choosing the compression algorithm of said plurality of compression algorithms having the highest compression rate.

19. The method of claim 18 further comprising the step of iteratively selecting said compression algorithm having the next lower compression rate until said ratio approaches one.

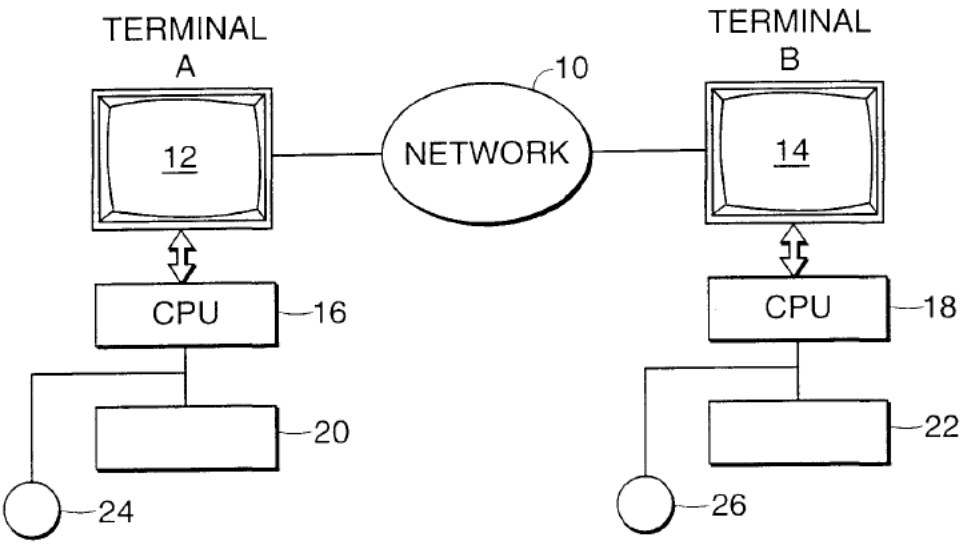


FIG. 1

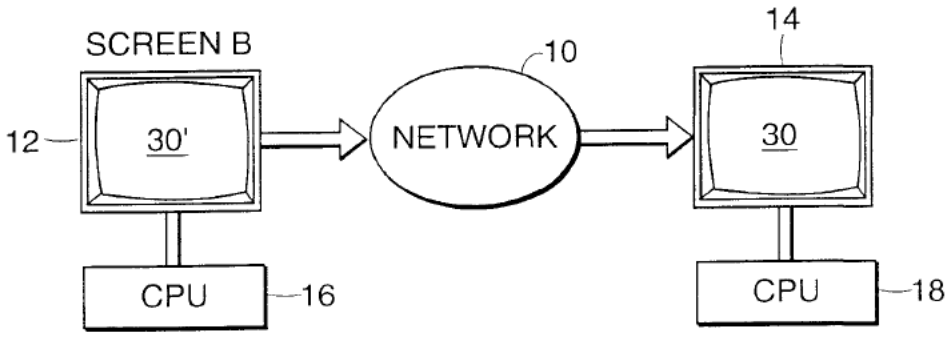


FIG. 2

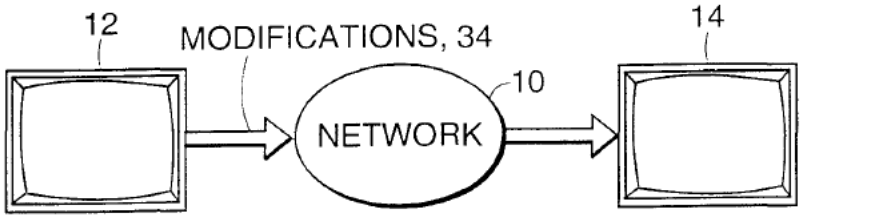


FIG. 3

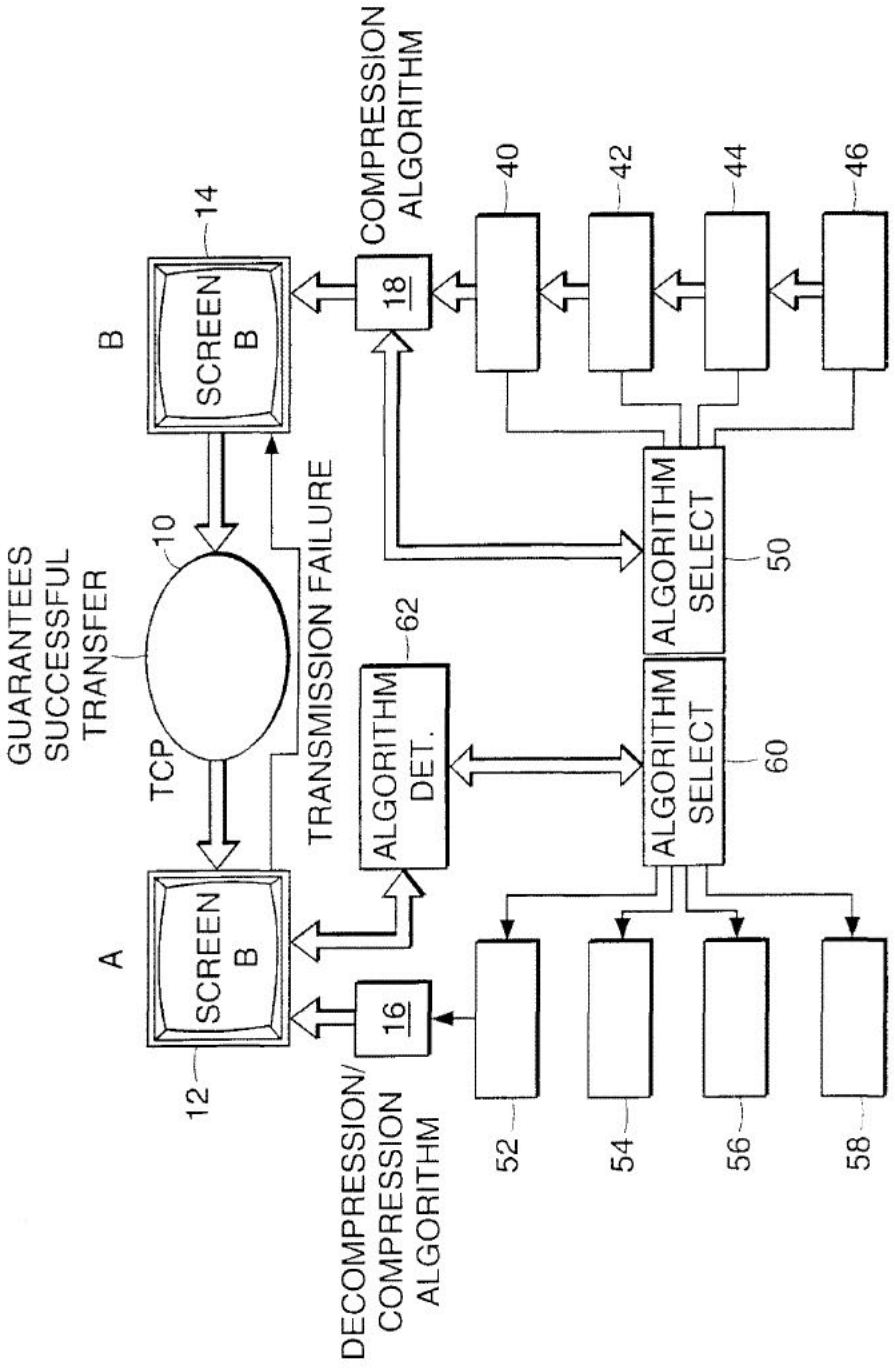

GUARANTEES SUCCESSFUL TRANSFER
TCP
10
A
SCREEN B
12
B
SCREEN B
14
TRANSMISSION FAILURE
DECOMPRESSION/ COMPRESSION ALGORITHM
16
COMPRESSION ALGORITHM
18
ALGORITHM DET.
62
ALGORITHM SELECT
60
ALGORITHM SELECT
50
40
42
44
46
52
54
56
58

FIG. 4

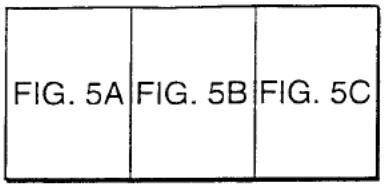


FIG. 5

```
#DEFINE CPX_CUSTOM_RLE          0X01
#DEFINE CPX_HUFFMAN_RLE         0X02
#DEFINE CPX_CRUSHER_RLE_9       0X04
#DEFINE CPX_CRUSHER_RLE_13      0X08

#DEFINE VIDEO_PAINT                          0X01
#DEFINE VIDEO_NO_PAINT                       0X02
#DEFINE VIDEO_CLOSE_CONNECTION               0X03
#DEFINE VIDEO_PAUSE                          0X04

STRUCT InfoBlock
{
        long cbCompressedSize;
        long cbFullSize:
        long nDirtyCount;
        DWORD fCompression;
        DWORD fStatus;
        DWORD fCommands;

        // UTILITIES
        InfoBlock ( )
        {
            Clear ( ) ;
        }
        void Clear ( )
        {
            cbCompressedSize = 0;
            cbFullSize = 0;
            nDirtyCount = 0;
            fStatus      = 0;
            fCommands = 0;
        }
        enum { PALETTE_AVAIL = 0X01 };
};

struct Status
{
        Status ( )
        {
            fStatus = 0;
        }
        void SetPause ( )
        {
            fStatus |= PAUSE;
        }
        void SetRefresh ( )
        {
            fStatus |= REFRESH;
        }
        bool Refresh ( )
        {
            if ( fStatus & REFRESH )
```

FIG. 5A

```
#DEFINE CPX_CUSTOM_RLE          0X01
#DEFINE CPX_HUFFMAN_RLE         0X02
#DEFINE CPX_CRUSHER_RLE_9       0X04
#DEFINE CPX_CRUSHER_RLE_13      0X08

#DEFINE VIDEO_PAINT                          0X01
#DEFINE VIDEO_NO_PAINT                       0X02
#DEFINE VIDEO_CLOSE_CONNECTION               0X03
#DEFINE VIDEO_PAUSE                          0X04

struct InfoBlock
{
        long cbCompressedSize;
        long cbFullSize:
        long nDirtyCount;
        DWORD fCompression;
        DWORD fStatus;
        DWORD fCommands;

        // UTILITIES
        InfoBlock ( )
        {
            Clear ( );
        }
        void Clear ( )
        {
            cbCompressedSize = 0;
            cbFullSize = 0;
            nDirtyCount     = 0;
            fStatus         = 0;
            fCommands = 0;
        }
        enum { PALETTE_AVAIL = 0X01 };
};

struct Status
{       Status ( )
        {
            fStatus = 0;
        }
        void SetPause ( )
        {
            fStatus = PAUSE;
        }
        void SetRefresh ( )
        {
            fStatus |= REFRESH;
        }
        bool Refresh ( )
        {
            IF ( fStatus & REFRESH )
```

FLAGS

ALGORITHM SELECTOR USING FLAGS

FIG. 5B

```
                return true;
            else return false;
        }
        bool Pause  (  )
        {
            if  ( fStatus & Pause )
                return true;
            else return false;
        }
        void Clear (  )
        {
            fStatus  = 0;
        }
        DWORD fStatus;
        enum { Pause = 0X02, REFRESH = 0X04 };
};

struct DirtyBlock
    short xPos;
    short yPos;

    // UTILITIES
    Void Mark ( int X, int Y)
    {
        xPos = ( short ) x;
        yPos = ( short ) Y;
    }
};
```

FIG. 5C

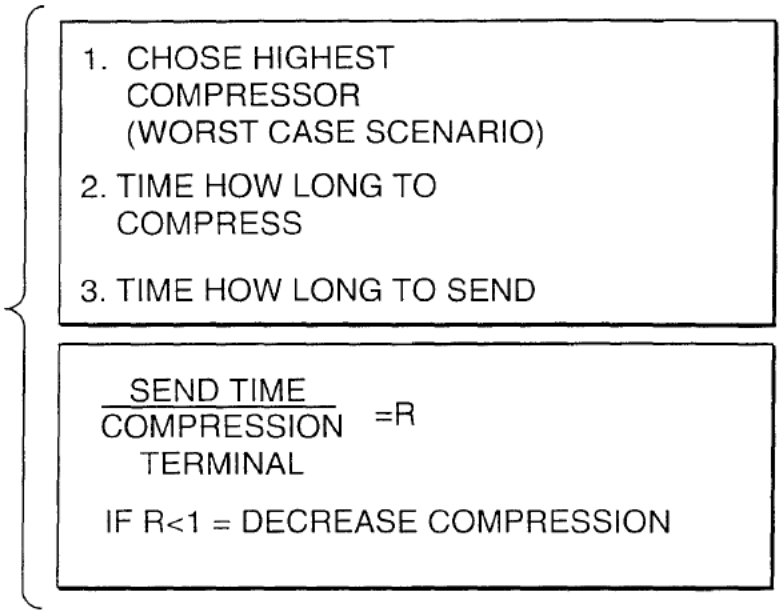

1. CHOSE HIGHEST COMPRESSOR (WORST CASE SCENARIO)
2. TIME HOW LONG TO COMPRESS
3. TIME HOW LONG TO SEND
SEND TIME / COMPRESSION TERMINAL =R
IF R<1 = DECREASE COMPRESSION

FIG. 6

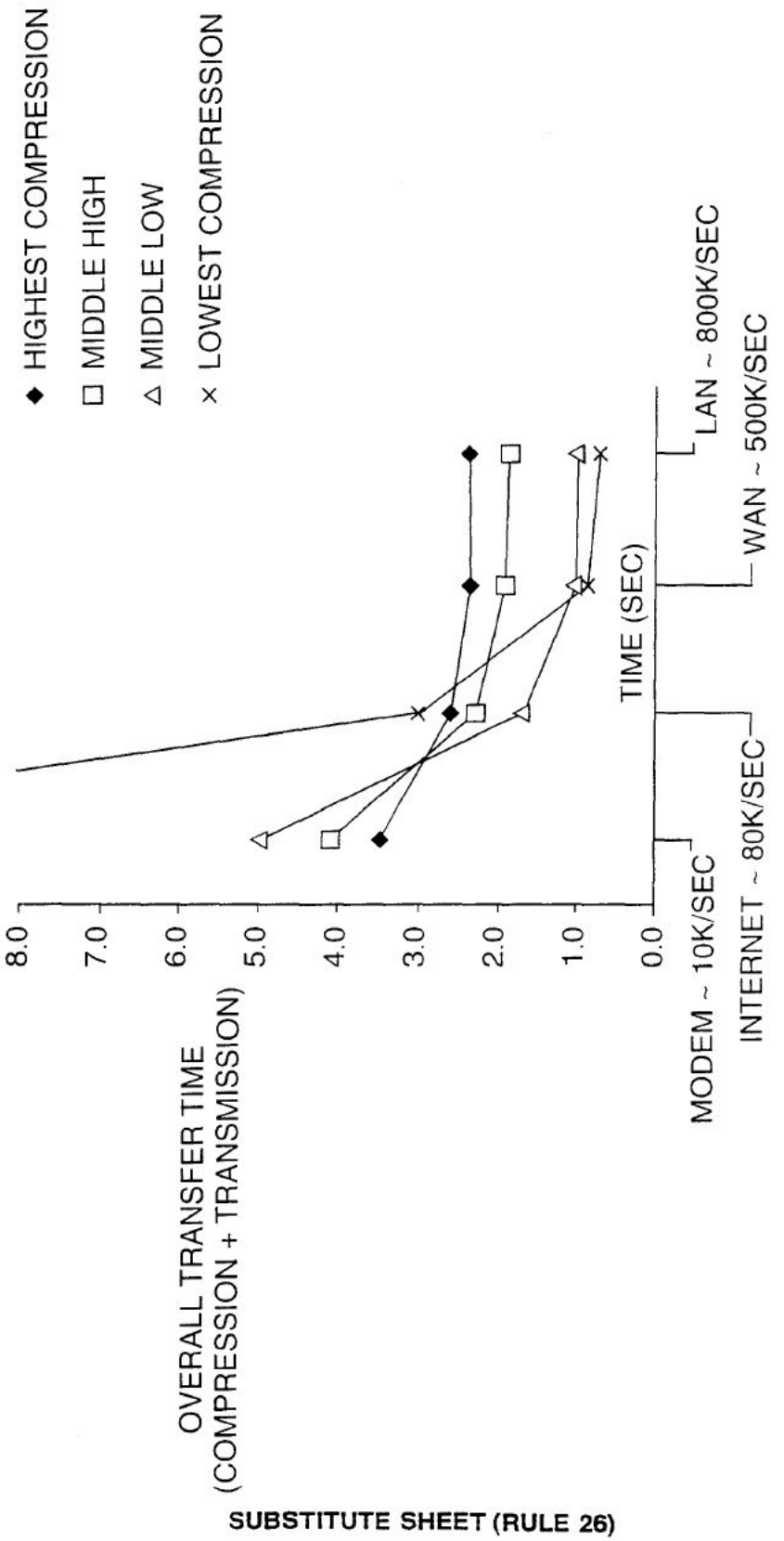

◆ HIGHEST COMPRESSION
□ MIDDLE HIGH
△ MIDDLE LOW
× LOWEST COMPRESSION
OVERALL TRANSFER TIME
(COMPRESSION + TRANSMISSION)
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
TIME (SEC)
MODEM ~ 10K/SEC
INTERNET ~ 80K/SEC
WAN ~ 500K/SEC
LAN ~ 800K/SEC

FIG. 7

INTERNATIONAL SEARCH REPORT

Inter ›nal Application No
PCT/US 98/24342

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F3/14

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F H03M

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	EP 0 468 910 A (IBM) 29 January 1992 see page 3, line 17 - line 31 see page 4, line 15 - page 6, line 11 see claim 14	1,12,17
A	FR 2 672 707 A (COMMANDE ELECTRONIQUE) 14 August 1992 see page 2, last paragraph see page 5, line 32 - page 10, line 19	1,5,12, 15,17
A	GB 2 296 114 A (IBM) 19 June 1996 see page 4, last paragraph see page 7, line 12 - page 11, line 6 see page 13, paragraph 2	1,12,17

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
11 March 1999	19/03/1999

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Amian, D

1

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

Inter nal Application No
PCT/US 98/24342

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0468910 A	29-01-1992	US 5276898 A	04-01-1994
		DE 69115152 D	18-01-1996
		DE 69115152 T	20-06-1996
		JP 2025136 C	26-02-1996
		JP 6069964 A	11-03-1994
		JP 7061086 B	28-06-1995
FR 2672707 A	14-08-1992	NONE	
GB 2296114 A	19-06-1996	DE 69506914 D	04-02-1999
		EP 0797796 A	01-10-1997
		WO 9618943 A	20-06-1996
		JP 10500514 T	13-01-1998

Form PCT/ISA/210 (patent family annex) (July 1992)