WIDAS MULTIMEDIA

COMPONENT VIEW REPORT

C++ SYNTAX

INCLUDES DOCUMENTATION

GENERATED AT MARCH 11, 2003 (VER 1.0)
MODIFIED AT JUNE 10, 2003 (VER 1.5 AND VER 1.5.1)

DOCUMENTED BY DONG HYUN JEONG

TABLE OF CONTENTS

CHAPTER 1. DEFINITIONS **5**

MULTI (WIDAS MULTIMEDIA) 5

CHAPTER 2. PROCEDURES **7**

AUTOMATIC PROCESSING 7
SELECTIVE PROCESSING 7
REPLAYING MEDIA 8
IDLE MONITORING 8
UPDATING UI PARAMETERS 9
SHOWING MESSAGE 9
AUTOMATIC SYSTEM SHUT-DOWN 10

CHAPTER 3. SOURCE CODE **11**

CMULTIAPP 11
CMAINFRAME 12
CCHILDFRAME CLASSES 18
CH26LCHILDFRMVIDEO 18
CH26LVIDEODOC 18
CH26LVIDEOVIEW 18
CH26LCHILDFRMAUDIO 18
CH26LAUDIODOC 18
CH26LAUDIOVIEW 18
CWAVELETCHILDFRMVIDEO 18
CWAVELETVIDEODOC 18
CWAVELETVIDEOVIEW 18
CWAVELETCHILDFRMAUDIO 19
CWAVELETAUDIODOC 19
CWAVELETAUDIOVIEW 19
CMP4CHILDFRMVIDEO 19
CMP4VIDEODOC 19
CMP4VIDEOVIEW 19
CMP4CHILDFRMAUDIO 19
CMP4AUDIODOC 19
CMP4AUDIOVIEW 19
CCHILDFRMRF 19
CQOSRFDOC 19
CQOSRFVIEW 20
CSAVECONTDLG 20
CSAVEONCEDLG 20
CABOUTDLG 20
CSTATUSDLGBAR 21
CSTATUSDLG 21
CSTATUSAUDIO 21
CSTATUSVIDEO 21
CSTATUSRF 21
MEDIA CLASSES DERIVED FROM EACH MEDIA DLLS 22
CMEDIAVAROPLAYER 22
CMEDIAH26LPLAYER 22
CMEDIAWAVELETPLAYER 23
CVAROPLAYER 23
CMWPLAYER 23

CHttpProtocol 23
CRasAPI 24
CControlPort 24
CPowerSerialPort 24
CRfSerialPort 24
CSvrIpDlg 24
CVideoScreenDlg 24
CMsgDlg 25
CTRACEMsgDlg 25
DEFINEd sdk 25
CSplashWnd 28
CPicture 28
CStaticLabel 28
CBufferStruct 28
CXPStyleButtonST 28
CTextRotator 28
ProgressBar 28
CToolBarEx 28
CExCheckBox 28
CGradientStatic 28
CLogFont 28
CItemBitmap 28
CEditMask 28
CControlToolTip 29
CColorListBox 29
COleFont 29
CRasConnectionDlg 29
CColorFormView 29
CArray<CTabItem*,CTabItem*> 29
CExStatic 29
CArray<CEXPropertyPage*,CEXPropertyPage*> 29
CRowCursor 29
CMSFlexGrid 29
CEXPropertySheet 29
CEditLabel 29
CButtonST 29
CControlPos 29
CAutoProcessDlg 29
CExToolbarWnd 30
CEXPropertyPage 30
CBitmapMenu 30
CThemeHelperST 30
CXPButton 30
CDockBarEx 30
CEXTabCtrl 30

APPENDIX A. DEFINED PACKET 31

APPENDIX B. RAS ARCHITECTURE 33

APPENDIX C. DATA STRUCTURE 35

APPENDIX D. DATA FLOW DIAGRAM 37

APPENDIX E. ACCESSING THE HTTP PROTOCOL 41

APPENDIX F. POWER CONTROLLER & IPC..43

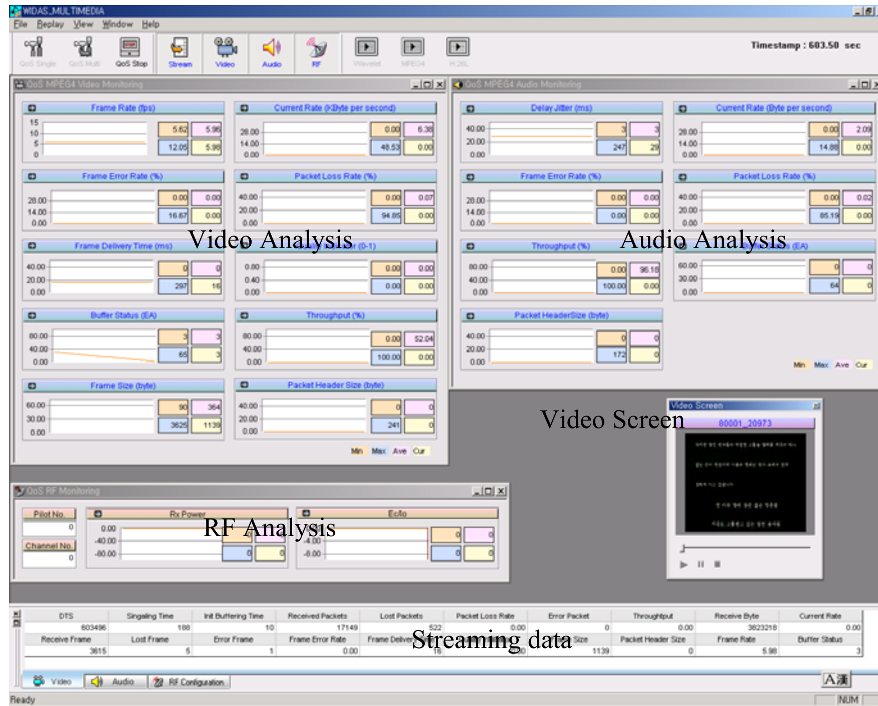
Chapter 1. Definitions

Multi (Widas Multimedia)

This software is for controlling three types of media. Those media are playing in June (Commercial brand name pronounced by SKT). The main purpose of this software is maintaining and evaluating quality of service (QoS) while using on demand services. It is a self-operation system which acts automatically after setting some parameters. Mainly it consists of system software and system device (called power control device). We exclude the system device, since it is not focus on this document. The focus of this documentation is that how the system software is designed and how it works automatically. Mainly the documentation consists of four parts such as Definitions, Procedures, Source code, and Appendix. Here in Definitions part, we will see what the system is and the notation method of source code. In Chapter 2. Procedures, we can see the brief procedures or scenarios used in the system. We also can see the technical information about source code of system in Chapter 3. At last, we can find some additional information in Appendix.

<Notation method>
All variables are named using Hungarian notation method. What is the Hungarian notation? It is a variable naming convention that includes C++ information about the variable in its name (such as data type, whether it is a reference variable or a constant variable, etc).

Briefly narrow some variables here (all variables are lower case character):
Class Member variable named with prefix m_.
Global variable named with prefix g_.
DWORD : dw
Double or DOUBLE: d or f
float : f
Int or INT : n
bool or BOOL : b
TCHAR or CString : sz or str
Pointer variables : p
Structure : st
Ex) m_pstH26l : member function, pointer and structure variable named with H26l


Video Analysis
Audio Analysis
Video Screen
RF Analysis
Streaming data
Timestamp : 603.50 sec

Chapter 2. Procedures

Automatic processing

As commented in CH 1., the main purpose of this software is running and evaluating QoS (Quality of Service) automatically without user's helps or commands. We called this Auto-processing. The Auto-processing denotes that if the OS or system power is shut down, the power is on by control device and operates the software to analyze media quality. We will skip the whole story about the power control device because this documentation is just for methodology of software implementation and modification. Anyway, is it possible to evaluate processing medium automatically? Yes, it is. First of all, user has to define evaluating parameters to enable automatic processing. After defining some parameters, the application can detect already defined parameters and operates by itself even if the system re-boot. How the application detects defined parameters? It uses a specified file which includes defined parameters (~$Multi.atx). The extension of the file stands for automatic execution. Whenever the application is started, it checks file existence. If there is, it starts automatic processing. It needs to have some extra time (twenty seconds) for checking components and changing routing table in advance.

Selective processing

Selective processing is a processing procedure that user can define parameters and select medium to process. We defined two different processing mode such as QoS Single and QoS Multi. The former is playing a media once. The latter is playing selected medium continuously until when user pushes the stop button. Whenever media plays, the application requests some parameters such as evaluating Quality Indicator (QI) and saving streaming data and log file to a local hard disk. Those are optional. If we want to evaluate QI, we must have an original media file to compare with streaming data. Probably you are curious to know why saving streaming data needs. It is used for replaying. We will have a look Replaying media briefly in next part.

What is the Selective processing does? Selective processing has several functions. To check system and software, there are some monitoring procedures such as media monitoring, http monitoring, serial-port monitoring and idle status monitoring. Media monitoring checks the media status while playing a media. If some error or stop messages are occurred, it catches the messages and sends them to other functions to process. Results or error messages will be sent to web server using Hyper Text Transfer Protocol (HTTP). While sending them, UI cannot know the network connection is alive or not. If it is broken, we have to wait the protocol returns timeout message (after 15 seconds), thus http monitoring is necessary for checking the network status. With serial port communication, power control device (designed by HFR co., Ltd.) can be controlled which supply system power. Sending and receiving messages synchronously between power control device and application are the key point of maintaining system. There are some unknown bugs in their application. If unknown bugs are occurred, the control device will shutdown the system and re-boot operating system.

There is an unwanted bugs that although application sends a playing request to media DLLs, the media does noting like idle status. Whenever the application gets into the idle status, we have to check and leave to get out of the idle status. The application checks the system is idle or not using timer function. Have a look Idle processing part to see more.

RF data can be addressed by sending requests through serial port. In Qualcomm documentation, there are some masking method for getting data from mobile. Brief description about addressing and getting data can be found in next chapter.

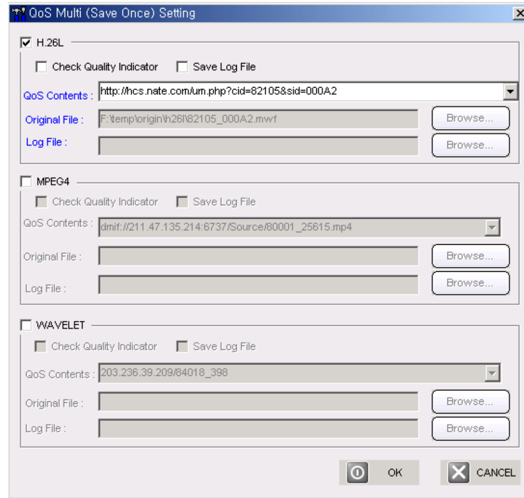


Figure 1. QoS Single

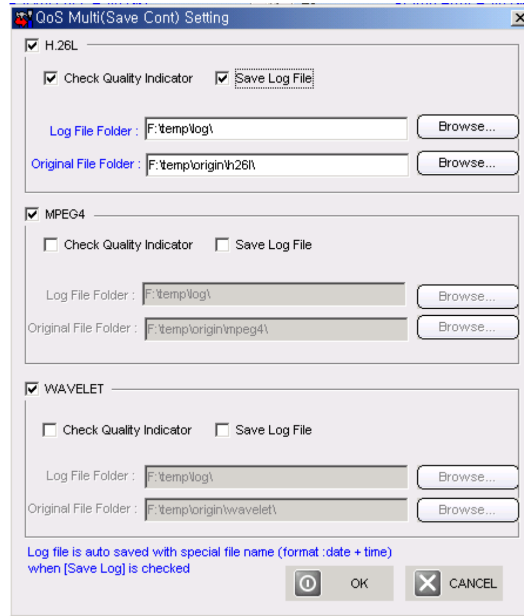


Figure 2. QoS Multi

Replaying media

As you understand it by name, the Replaying media is to replay saved data. When replaying a saved media, maintaining synchronization is important between media data and a log file which includes evaluated data. To manage synchronization efficiently, it loads evaluated data in advance. But the most serious one is that user moves the sliding bar to backward or forward position. If so, searching synchronized position in streaming data and log file needs robust time consumption. To minimize searching time, it uses pre-processing procedure which minimize the robust time. Internally if user requests skipping commands (moving backward or forward), application sends a pause message with changed synchronous position, and moves the sliding bar with changed position and sends resume or play message.
In Mpeg4 media, it has to send stop message to media DLL after finishing replay because of the architecture of DLL (commented by VaroVision Inc.).

Idle Monitoring

As we commented shortly in Selective processing part, Idle Monitoring is indispensable for checking the system. While playing medium in multi-processing mode, the application can go through to idle status intermittently. To detect this error, it uses update timer function with pre-defined time value

(SERVERTIMEOUT and MEDIAIDLETIME). There are two different idle checking procedures such as checking media DLLs and the application.

First, the application checks DTS time. If DTS of streaming data is updated or not. It regards the first incoming data as a start time. If the streaming data does not come out for the pre-defined time (SERVERTIMEOUT) comparing with the start time, it treats the media DLL as idle. Therefore media will be stopped and send Error message (202) to web-server.

Second, the application checks sending messages. If error or result messages are not sent to web-server for the pre-defined time (MEDIAIDLETIME), it regards as idle status; thus reset the system without sending error message to web-server.

Updating UI parameters

Decoded parameters will come out while playing it. Each data depend on each medium; therefore updating intervals are different. Even if parameter passing interval time is defined, decoding time depends on each decoders; therefore, there are two different updating modes such as active (direct) and passive updating. Passive updating means that evaluated parameters will be saved temporally in a specified memory, system acts as a substituting procedure; therefore, updating it in specified intervals (MULTI_UPDATEMONITORINGTIMER). The other one is active (direct) updating. In this mode, decoded data will be updated whenever it comes out.

Decoded data has to be changed. With raw data, user (Administrator) can not understand. It has to be changed to easily recognizable form. After processing, it can be shown as a chart and static data.

Important: Even if we defined passive mode which uses time interval, the processed data will be saved to log file following active mode pattern. In replaying mode, as video and decoded data can be synchronized using DTS (data time stamp), it is hard to find exact time stamp position with altered DTS if application saves modified DTS to log file.

Showing Message

After the system is re-booted, we can see the message that the automatic processing procedure is in process; furthermore, whenever unwanted error is occurred, the system shows the error message. All are for letting user (Administrator) know the system status. With the displayed message, administrator can recognize the status of the system. It uses a timerfunction, since while showing message without time (how long the message will be posted), user cannot can realized it is acting or not. Message showing time will be decreased in a second. After the defined time is elapsed, message dialog will be disappeared by itself. The elapsed times are defined differently depends on each message attributes such as mobile reset time, automatic processing time, and etc. See more about defined time in “define.h”.

Automatic system shut-down

While playing medium (multi-processing mode), system will be shut-down by itself between 2:00AM ~ 2:30AM(default). It is the optional procedures. If unknown error is occurred in software or operating system, application cannot recognize the system has to be rebooted. It can be changed the default shut-down value in “define.h”.

Chapter 3. Source code

CMULTIAPP

//
CMultiApp:
See Multi.cpp for the implementation of this class
Derived from CwinApp

<Screen Changing>
Listed items used for switching screen depends on each media type. Whenever user selects some media to play, those functions will be called. Therefore user can see different screen depends on each media.

```
CMP4ChildFrmVideo*          m_pMP4VideoChildFrame;
CMP4ChildFrmAudio*          m_pMP4AudioChildFrame;
CMP4VideoView*              m_pMP4VideoView;
CMP4AudioView*              m_pMP4AudioView;
CMultiDocTemplate*          m_pMP4VideoDocTemplate;
CMultiDocTemplate*          m_pMP4AudioDocTemplate;

CWaveletChildFrmVideo*      m_pWaveletVideoChildFrame;
CWaveletChildFrmAudio*      m_pWaveletAudioChildFrame;
CWaveletVideoView*          m_pWaveletVideoView;
CWaveletAudioView*          m_pWaveletAudioView;
CMultiDocTemplate*          m_pWaveletVideoDocTemplate;
CMultiDocTemplate*          m_pWaveletAudioDocTemplate;

CH26LChildFrmVideo*         m_pH26LVideoChildFrame;
CH26LChildFrmAudio*         m_pH26LAudioChildFrame;
CH26LVideoView*             m_pH26LVideoView;
CH26LAudioView*             m_pH26LAudioView;
CMultiDocTemplate*          m_pH26LVideoDocTemplate;
CMultiDocTemplate*          m_pH26LAudioDocTemplate;

CChildFrmRf*                m_pRfChildFrame;
CQOSRFView*                 m_pRfView;
CMultiDocTemplate*          m_pRFDocTemplate;

CONPARAMETER                m_ConPameter;   // temporary save used for reference item
```

<Monitoring idle status of Media>
While playing media, main monitoring functions defined in MainFrame(Multi_MonitoringMedia(…) and Single_MonitoringMedia(…)) can not know the status of media when the media goes to idle. Listed functions do check media is stopped or goes to idle status. How to check the status of media? After time-out value is specified (SERVERTIMEOUT which is defined in “define.h”), and every time a time-out occurs, the system posts a message to application-defined TimerCheckMediaDllProc() callback function. This call-back function will process output stream (parameter) which comes from Media DLLs. If the DTS does not updated for time-out value, it will be considered the status of Media as Idle. Therefore media will be stopped and send the defined error messages (202 or 203) to web.

```
UINT m_nCurrentDts;      // current DTS value
UINT m_nOldDts;          // old DTS value
UINT m_nTimer;           // timer identifier used to kill the timer
static  void CALLBACK TimerCheckMediaDllProc();
UINT StartCheckMedialDllMonitoring();
void   StopCheckMedialDllMonitoring();
```

<Updating parameter>
Receiving parameter must be updated whenever streaming data comes out. Original parameter cannot be updated immediately because it has some unknown or unwanted information. Therefore it is refined in API function (WAPI_ProcessRealtimeData(…)), then update function(UpdateData(…)) will be called shown below. While updating refined data, the function will do three kinds of things such as counting, checking and updating. To calculate average value of each parameter it is necessary to count updated times. Received data must be refined in API function, but RF data doesn't. So RF data will be checked here that the data is correct or not. And update refined data and RF data.

```
UINT m_nTotal;           // count used for calculate average value for each parameter
void   UpdateData(CONPARAM *pParam, CONPARAMSTR *pString);
```

<Updating parameter with interval time>
As we saw the updating procedure in Chapter 2, here are the function which update evaluated parameters in predefined time. Here can see the callback function (UpdateMonitoringMediaProc). Whenever updating is in progress, it checks the new updated parameter is exists or not. If it is not updated, it updates with Null parameter to indicate the system is acting but media parameter does not come out.

```
UINT m_nUpdateMonitoringMediaTimer;
BOOL m_bUpdateData;
static void CALLBACK UpdateMonitoringMediaProc();
CCriticalSection m_csUpdateData;
void StartUpdateMonitoringMedia();
void StopUpdateMonitoringMedia();
```

<Getting and Setting statusbar pointer>
The Windows Forms **StatusBar** control is used on forms as an area, usually displayed at the bottom of a window, in which an application can display various kinds of status information. In this application we used to indicate predecoding status of media file. But the problem is that media classes cannot access directly to StstuaBar therefore we save StatusBar control temporary. With saved StatusBar control media can access StatusBar control and show predecoding status.

```
CStatusBar*  m_pStatusBar;
CStatusBar*  GetStatusBar();
void SetStatusBar(CStatusBar* pStatusBar);
```

<Setting media start time and end time>
System sends evaluated media parameter to web-server with media start time and end time. Also the media start time is used to checking the idle time. If the system does not act for a long time, it cannot recognized itself. It is opposite with self-operational system.

```
SYSTEMTIME SetMediaStartTime();
void SetMediaEndTime();
```

CMAINFRAME

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

Derived from CMDIFrameWnd

All components in the application are divided into two groups such as managing and controlling. Managing part has some sub parts : QoS Single, QoS Multi and Replaying media. While playing a media, each part can be used for playing and controlling media status (active, stop or error occurred). QoS single and Multi denote real-time playing media method. Replaying has extra functions : pause, stop and move forward or backward.

And controlling group has several thread functions and control classes such as RF Serial, Control device Serial and Http control class. Also there are a lot of flags used to check the status of each control class.

Managing groups are Auto-Processingthread, Single-Monitoringmedia, Multi-Monitoringmedia and Replay-Monitoringmedia. Those functions always check the status while playing media. If message occurred by Media components (H26L, MPEG4 and WAVELET), those functions can detect the message. When error or finish message occurred, Multi-Monitoringmedia stops playing media, and check the other media type and items. And then it starts playing another media. Otherwise in Single-Monitoringmedia will stop playing and monitoring. Replay-Monitoringmedia does like Single-Monitoringmedia.

<Auto-Processing>

Auto-processing is to run automatically as soon as program started. It read a file “init.txt” to load RF mode and web-server IP address. And then check "~$Multi.atx" is exist or not. The file has setting values (User selected options). If the file exists, AutoProcessingThread function call OnAutoProcessingSavecont() function to play medium with the setting values.

```
CWinThread *m_pAutoProcessingThread;
BOOL        m_bAutoProcessingEnable;// Auto processing check bit
static      UINT AutoProcessingThread(LPVOID pParam);
void        OnAutoProcessingSavecont();
```

<HTTP controlling >

It is for sending error or result data to web-server. When some error is occurred, monitoring groups post message to send result or error data. The media (H.26L) needs to pass authentication process. It connects to HTTP server to get redirect URL with some data. It is designed using thread functions because the GUI does not know the network status (stable or not). When the network is unstable, it cannot connect to HTTP server directly then error will be occurred after 15second. We use here thread functions. The thread is to update UI resources when the network is unstable. Certain UI resources should only be accessed on the same thread, on which they were created. Therefore while the network connecting process holds onto OS resources, other UI process can access OS resources when we use thread function.

```
CString        m_strMeasureStartTime;  // Media playing start time
UINT           m_nSendMsgStatus;
UINT           m_nGetRedirectUrl;
CHttpProtocol  m_HttpProtocol;
CString        m_strH26LSelectedUrl;
CString        m_strH26LRedirectUrl;
UINT           m_nErrMsg;
BOOL           m_bErrorOccurred;
BOOL           m_bSendResultToWeb;          // Send evaluation result to web server
CWinThread *m_pGetRedirectUrlThread;
CWinThread *m_pSendErrorToWebThread;
CWinThread *m_pSendMsgToWebThread;
void           GetRedirectUrl(URLSTRUCT *stUrl, UINT nMode);
BOOL           ConnectHttp(CString *pstrUrl, CString *pRedirectUrl, UINT nMethod);
BOOL           SendResultToWeb();
```

```
BOOL        SendErrorToWeb(UINT nMsg);
static      UINT GetRedirectUrl(LPVOID pParam);
static      UINT SendErrorToWeb(LPVOID pParam);
static      UINT SendMsgToWeb(LPVOID pParam);
CString     CalcSendMsg(UINT nErrorCode = NULL); // Calculate msg
```

<QoS Single and QoS Multi>

To play a media, functions listed below are used. Single denotes playing a media and Multi is for playing selected medium with predefined data listed in files (“mpeg4.txt”, “wavelet.txt” and “h.26l.txt”) continuously. While playing media, some unwanted errors might be occurred. When it happens, monitoring thread functions check the error type and send messages to other functions to solve the problem. QoS Single and Multi monitoring thread function has a bit different processing procedure. When error occurred while playing media, the former function checks it and sends error messages to web server and posting a message (WM_MEDIAASSERT) to stop playing media. Otherwise the latter is different. Checking error is the same as the former one but when error occurred, it sends error message to web and checks next media to play continuously.

Scenario of QoS Single play :

When user pushed QoS Single button, OnMenuitemSaveonce() function called. The function create modal dialog (CSaveOnceDlg) to get setting values (Media type, Qi and Log save). After selected by user, it checks that what kind of media type is selected and Qi or Log save item is. After finished checking selected options, the application posts a message (WM_PLAYREALTIMEMEDIA) to play media and start Monitoring.

```
// QoS Single Play related items
CString     m_strLogFileName;
CString     m_strDataFileName;
int         m_nSaveMethod;
int         m_nSelectedType;        // Which type is selected when Save button is pressed
BOOL        m_bSaveDataFile;
int         m_nSinglePlayingMedia;  // position of playing media
CString     m_strLogFilePath;
PROCESSSINGLE m_stProcessSingleSel;
static      UINT Single_MonitoringMedia(LPVOID pParam);
```

Scenario of QoS Multi play :

QoS Multi play is more complicate than QoS Single. It has many flags and related functions because it has to check media status and play medium continuously until user pushed stop button. When user pushed QoS Multi button, OnMenuitemSaveCont() function called. The function create modal dialog (CSaveContDlg) to get setting values (Media type, Qi and Log save). After selecting, it checks that what kind of media type is selected and Qi or Log Save item is checked. After finished checking user options, it saves setting values to a file (~$Multi.atx) and starts Monitoring in sequence. In QoS Multi play, playing message (WM_PLAYREALTIMEMEDIA) is occurred in Monitoring function. It is different from QoS Single play. Monitoring function is divided into two main parts to check media briefly. One is checking Media stop or cancel message. When user canceled or stopped a playing media, monitoring function has to act immediately so it must check the playing media all the time. The other is checking RAS connection. When RAS is connected, it sends playing message to media, otherwise it sends stop message. QoSMulti must be passed OnMulti_ChkMedia() and QoSMulti()function. The former is for checking what kinds of media are set and the latter is for checking media parameters. In QoSMulti() function, playing message will be generated.

```
// QOS Multi Play related items
```

```
CString         m_strLogFilePathH26l;
CString         m_strLogFilePathMpeg4;
CString         m_strLogFilePathWavelet;
int             m_nMultiPlayingMediaMpeg4;
int             m_nMultiPlayingMediaWavelet;
int             m_nMultiPlayingMediaH26l;
BOOL            m_bCheckH26l;
BOOL            m_bCheckMpeg4;
BOOL            m_bCheckWavelet;
PROCESSMULTI m_stProcessMultiSel;
void            QoSMulti(int nPlayingMedia, UINT nMode);
static          UINT Multi_MonitoringMedia(LPVOID pParam);
```

<QoS Replay>

Saved log file and media file can be replayed. It also has a Monitoring function. The key feature of replaying media is that evaluated value in the logfile data and media timestamp value must be synchronized. See VideoScreenDlg class to see brief explanations.

```
// QoS Replay related items
static          UINT Replay_MonitoringMedia(LPVOID pParam);
void            ReplayMedia(UINT nMedia, CString strMediaFileName);
```

<Log File>

It is used to read or write evaluated values in a file. It has a check bit(LOGFILECHKBIT) which indicates the log file generated by Widas Multimedia Software.

```
//      Logfile related items
BOOL            m_bFileOpen;
CFile*          m_pLogFile;
BOOL            OpenLogFile(CString strLogFileName, BOOL bMode = TRUE);
void            CloseLogFile();
```

<RF Data>

There are two kinds of RF mode in here. One is EVDO and the other is 1X mode. To get RF data, we have to send a request through serial port. The brief explanation how to get RF data can find in QualComm (***CDMA Dual-Mode Subscriber Station Serial Data Interface Control Document)*** documentation. When we want to get EVDO RF data, we just send a request once. But in 1X mode, we have to send continuously if we want to get data. In EVDO, is it restricted sending a message once? No. You don’t need to worry about sending requests several times. We used a timer function here to send a request message. Whenever set time elapsed, timer function will send a request message through a serial port. The important thing here is that in EVDO mode RF data comes out continuously but 1X mode does not. If we want to evaluate EVDO mode RF data, we have to control update routine. This means that if RF data updates whenever it comes out, updating resource hold OS resources therefore other OS resource cannot be used. We restricted a point of updating time to the streaming data with timestamp value. Also we need to check what the most frequently displayed value is (Pilot No., Channel No.) and send them to web server because we cannot see all data on the Internet.

```
// RF Data
RFSTRUCT        *m_pRfPilotNoListHead;
RFSTRUCT        *m_pRfChannelListHead;
RFPARAM         m_Rfparam;
CCriticalSection m_csRfData;
BOOL            m_bRFMonitoring;
BOOL            m_bRfparamUpdate;
BOOL            m_bRFMode; // 0 : EVDO / 1 : 1x
```

```
void        InitRFData();
void        RefreshData();
void        CalcRfData(RFPARAM *pRfParam);
void        OnUpdateRealtimeRfData();
BOOL        OnRFLogRequest();      // Send RF data request
```

<Gateway Set >

When OS (Operating system) is initialized, the gateway will be set if we defined network devices. The RAS is also a kind of networking, therefore when RAS is connected, gateway (using 0.0.0.0) will be added with new host IP address. As you understand, this software is to evaluate media quality through mobile. If we did not define the initial gateway, we cannot guarantee that the media we evaluate received data through Mobile or Ethernet. So we have to remove Ethernet gateway (using 0.0.0.0) generated by initial OS networking. Moreover we have to add web-sever route. The web-server will be located in LAN area, therefore using Ethernet is much more useful than RAS.

Typically, a packet may travel through a number of network points with routers before arriving at its destination. In personal computer, routing table is used for reference. Therefore if destinations are not defined in routing table, applications will use default gateway whose destination IP address might be set to 0.0.0.0.

```
// Gateway
DWORD       m_dwLocalIp;
DWORD       m_dwWebServerIp;
CString     m_strWebServerIp;
void        InitGateway();
void        DeInitGateway();
PMIB_IPFORWARDROW       m_pIpForwardArrow;
```

Initialized Routing table (each values depends on the system)

```
===========================================================================
Interface List
0x1 ........................... MS TCP Loopback interface
0x3000003 ...00 e0 18 54 5a 7f ...... HP 10/100TX PCI Ethernet Driver
===========================================================================
===========================================================================
Active Routes:
Network Destination    Netmask            Gateway            Interface          Metric
        0.0.0.0        0.0.0.0            210.115.229.1      210.115.229.124    1
210.115.229.0          255.255.255.0      210.115.229.124    210.115.229.124    1
  210.115.229.124      255.255.255.255    127.0.0.1          127.0.0.1          1
  210.115.229.255      255.255.255.255    210.115.229.124    210.115.229.124    1
      224.0.0.0        224.0.0.0          210.115.229.124    210.115.229.124    1
  255.255.255.255      255.255.255.255    210.115.229.124    210.115.229.124    1
Default Gateway:    210.115.229.1
===========================================================================
Persistent Routes:
None
```

<Serial Port >

To control RF data and Power control device, we use serial port. In short, we had a look about how to get RF data through serial port. We will skip this, those functions listed below just do making serial port connection. See brief explanation in CRfSerialPort and CPowerSerialPort classes.

```
//  Serial Port
BOOL        m_bRfSerialPortConnection;
BOOL        m_bPowerSerialPortConnection;
BOOL        CreateRfSerialPortConnection();
```

```
BOOL            CreatePowerSerialPortConnection();
```

<System Shutdown >
Before the system shutdown, it send a message to power-serial port or to web-server. Therefore it have to wait for a while sending message.

```
// Shutdown program and system
BOOL            m_bWindowShutdown;
CThread         *m_pWindowShutdownThread;
static          UINT WindowShutdown(LPVOID pParam);
CString         m_strShutDownMsg;       // T3 or T5
```

CCHILDFRAME CLASSES

MFC Multiple document Interface (MDI) has a child window which consists of child frame, document and view. Listed items below are H26L, MPEG4, WAVELET and RF classes. Those have similar definitions. Each view class is used to update streamed data. Everyone knows that what kinds of class will be used if he studied MFC; therefore we will skip some classes.
What is a derived class? The derived class is a modified class originally defined in MFC basic to change or insert some functions. We used some derived class to change form and text color. Also we used ChartFX component to display real time graph and MSFlexGrid to make a possibility seeing all data at once.

CH26LCHILDFRMVIDEO

///
CH26LChildFrmVideo frame
Derived from CMDIChildWnd

CH26LVIDEODOC

///
CH26LVideoDoc document
Derived from CDocument

CH26LVIDEOVIEW

Derived from CColorFormView

CH26LCHILDFRMAUDIO

///
CH26LChildFrmAudio frame
Derived from CMDIChildWnd

CH26LAUDIODOC

///
CH26LAudioDoc document
Derived from CDocument

CH26LAUDIOVIEW

Derived from CColorFormView

CWAVELETCHILDFRMVIDEO

///
CWaveletChildFrmVideo frame
Derived from CMDIChildWnd

CWAVELETVIDEODOC

///
CWaveletVideoDoc document
Derived from CDocument

CWAVELETVIDEOVIEW

Derived from CColorFormView

CWAVELETCHILDFRMAUDIO

///
CWaveletChildFrmAudio frame
Derived from CMDIChildWnd

CWAVELETAUDIODOC

///
CWaveletAudioDoc document
Derived from CDocument

CWAVELETAUDIOVIEW

Derived from CColorFormView

CMP4CHILDFRMVIDEO

///
CMP4ChildFrmVideo frame
Derived from CMDIChildWnd

CMP4VIDEODOC

///
CMP4VideoDoc document
Derived from CDocument

CMP4VIDEOVIEW

Derived from CColorFormView

CMP4CHILDFRMAUDIO

///
CMP4ChildFrmAudio frame
Derived from CMDIChildWnd

CMP4AUDIODOC

///
CMP4AudioDoc document
Derived from CDocument

CMP4AUDIOVIEW

Derived from CColorFormView

CCHILDFRMRF

ChildFrmRf.h : header file

///
CChildFrmRf frame
Derived from CMDIChildWnd

CQOSRFDOC

///

CQOSRFDoc document
Derived from CDocument

CQOSRFVIEW

Derived from CColorFormView

CSAVECONTDLG

///
CSaveContDlg dialog

As we denoted in QoS Multi section, it is for selecting or changing media attributes. The key point of this class is that Logfile or Original file directories must be checked before using them because if the defined directories are illegal, media DLLs may occur some unwanted errors. Also changed attributes have to be saved in each media files for next usage.

```
void SaveUrlFile(URLSTRUCT *stUrl, UINT nMedia);
BOOL CheckDirectory(CString *szpPath);
```

CSAVEONCEDLG

///
CSaveOnceDlg dialog
It is a bit similar with CSaveContDlg, but the difference is that it is not possible to select several media types to play. Only one media type can be selected and also there is no saving routine. Instead of saving we maintain changed attributes.

```
void EnableH26lItems(BOOL bEnable = FALSE);
void EnableMpeg4Items(BOOL bEnable = FALSE);
void EnableWaveletItems(BOOL bEnable = FALSE);
void InitData(URLSTRUCT *pUrl, CComboBox *pComboBox);
```

CABOUTDLG

///
CAboutDlg dialog used for App About

CSTATUSDLGBAR

///
CStatusDlgBar window
Derived from CControlBar

Statusbar are composed of CStatusDlg which has also three pages to show video, audio and RF data. To display data, MsFlexGrid component are used for each dialog. And each dialog views can be switched using tab control button.

CSTATUSDLG

///
CStatusDlg dialog

This class is used as a base dialog of CControlBar. Mainly this class does not do anything. It acts as a key roll send stream data to each page (Video, Audio and RF). Each page uses MSFlexGrid to display received data.

```
CEXPropertySheet m_Sheet;                // tab control
CControlPos      m_cControlPos;          // to control window size
CStatusVideo     *m_pStatusVideoPage;    // Video page
CStatusAudio     *m_pStatusAudioPage;    // Audio page
CStatusRf        *m_pStatusRfPage;       // RF page
CBitmap          m_bmStatusIcon[3][2];   // tab control button BMP image
void UpdateData(CONSTRING *pString);
void UpdateData(RFPARAMETER *pParam);
```

CSTATUSAUDIO

///
CStatusAudio dialog
Derived from CEXPropertyPage

CSTATUSVIDEO

///
CStatusVideo dialog
Derived from CEXPropertyPage

CSTATUSRF

///
CStatusRf dialog
Derived from CEXPropertyPage

MEDIA CLASSES DERIVED FROM EACH MEDIA DLLS

As we have a look before, three are three different kinds of media DLLs (VaroPlayer.dll, TCM3QMSLib.dll and win32_player_dll.dll). Each media class includes some DLL functions or DLL classes. We defined some nested classes include DLL functions or DLL classes : CMediaH26lPlayer, CMediaVaroPlayer and CMediaWaveletPlayer. They have similar syntax and functions.

CMEDIAVAROPLAYER

This class is imported or exported from the VaroPlayer.dll

```
CVaroPlayer     m_VaroPlayer;
BOOL            m_bPlayerInit;          // Initialization is set or not
UINT            m_nMediaStatus;         // current media status
UINT            m_nMediaErrorCode;      // media error code
BOOL            m_bMediaStartBit;       // media started or not
UINT            m_nMediaPlayingMode;    // current media mode
BOOL            m_bPredecoding;         // Predecoding enabled
void            InitPlayer(HWND hWnd, HWND hWnd_display, char *PhoneNum);
BOOL            MediaPlay(char* szURL, UINT nSizeURL, BOOL bSaveflag,
                char* szSavefileName, char* szOrgfileName, CProgressCtrl *pProgress);
void            MediaPlayEnd();
void            MediaStop();
BOOL            MediaReplay(char* szFilename);
void            MediaReplayStop();
void            MediaPause(void);
void            MediaResume(void);
void            MediaMoveBegin(unsigned int timeToMove);
void            MediaMoveEnd(unsigned int timeToMove);
UINT            MediaGetContentDuration(char* szFilename);
static void     MediaDecodePutParam(unsigned int  DTS, MParameter *pParam);
static void     MediaEnd(unsigned int  reason);
static void     MediaPutQIProgress(unsigned int  Progress);
```

CMEDIAH26LPLAYER

This class is imported or exported from the win32_player_dll.dll

```
CMWPlayer       m_H26lPlayer;
BOOL            m_bPlayerInit;          // Initialization is set or not
UINT            m_nMediaStatus;         // current media status
UINT            m_nMediaErrorCode;      // media error code
BOOL            m_bMediaStartBit;       // media started or not
UINT            m_nMediaPlayingMode;    // current media mode
BOOL            m_bPredecoding;         // Predecoding enabled
void            InitPlayer(HWND hWnd, HWND hWnd_display);
BOOL            MediaPlay(char* szURL, UINT nSizeURL, BOOL bSaveflag,
                char* szSavefileName, char* szOrgfileName, CProgressCtrl *pProgress);
BOOL            MediaReplay(char* pszFilename);
void            MediaStop();
void            MediaPause(void);
void            MediaResume(void);
void            MediaMoveNMP(UINT timeToMove);
```

```
UINT            MediaGetContentDuration(char *pszFilename);
void            MediaPlayEnd();
static void     MediaDecodePutParam(UINT timeLastDecode, MParam *pParam);
static void     MediaEnd(int nReason);
static void     MediaPutQIProgress(INT Progress);
```

CMEDIAWAVELETPLAYER

This class does not have external classes. As you can see the difference between this and other media classes. There are original classes derived from DLLs such as CVaroPlayer and CMWPlayer, but here it isn't. Just some functions are exported from the TCM3QMSLib.dll

```
BOOL            m_bPlayerInit;          // Initialization is set or not
UINT            m_nMediaStatus;         // current media status
UINT            m_nMediaErrorCode;      // media error code
BOOL            m_bMediaStartBit;       // media started or not
UINT            m_nMediaPlayingMode;    // current media mode
BOOL            m_bPredecoding;         // Predecoding enabled
void            InitPlayer(HWND hWnd, HWND hWnd_display, char *PhoneNum);
BOOL            MediaPlay(char* szURL, UINT nSizeURL, BOOL bSaveflag,
                char* szSavefileName, char* szOrgfileName, CProgressCtrl *pProgress);
void            MediaPlayEnd();
void            MediaStop();
BOOL            MediaReplay(char* szFilename);
void            MediaReplayStop();
void            MediaPause(void);
void            MediaResume(void);
void            MediaMoveBegin(unsigned int timeToMove);
void            MediaMoveEnd(unsigned int timeToMove);
UINT            MediaGetContentDuration(char* szFilename);
static void     MediaDecodePutParam(unsigned int  DTS, MParameter *pParam);
static void     MediaEnd(unsigned int  reason);
static void     MediaPutQIProgress(unsigned int  Progress);
```

CVAROPLAYER

This class is exported from the VaroPlayer.dll
Mpeg4 Media defined by Component company

CMWPLAYER

This class is exported from the win32_player_dll.dll
H26L Media defined by Component company

CHTTPPROTOCOL

//
CHttpProtocol command target
To send processed data or error to web-server, we use this. Also H.26L media has redirecting url method using this class. There are two kinds of posting method in HTTP such as Get and Post method.

We tested the methods both and decided to send message using Get method. When we use Post method, we found that HTTP parser page didn't recognize sometimes (We don't know why?).

```
BOOL HttpConnectionCheck();
BOOL HttpConnection(CString szUrl, CString szHeaders);
BOOL HttpGetMessage(TCHAR *pszMsgResult, TCHAR *pszMsg);
BOOL HttpSendMessage(TCHAR *pszMsgResult, TCHAR *pszPostData);
BOOL HttpMakeConnection(LPCTSTR szAddress, LPCTSTR szHeaders);
BOOL HttpPostMessage(TCHAR *strRcvValue, LPCTSTR szAddress, LPCTSTR szHeaders,
        char *szPhoneNum, TCHAR *szPostData, UINT nMode, UINT nMethod );
static UINT MsgPostingThread(LPVOID pParam);
```

CRASAPI

CCONTROLPORT

///

CControlPort window of CRasAPi

To use RAS API, we have to define some arguments. This class is for communicating with RAS Api class.

```
DWORD InitRasAPI(LPCTSTR pstrStatus, LPCTSTR pstrModemName,
                 LPCTSTR pstrPasswd, LPCTSTR pstrPhoneNum,
                 LPCTSTR pstrUser, LPCSTR pstrEntry);
DWORD DialUpNetwork(LPVOID lpvNotifier = NULL);
BOOL DialHangUp();
BOOL ConnectRasAPI(LPVOID lpvNotifier = NULL);
BOOL DisConnectRasAPI();
```

CPOWERSERIALPORT

///

Controling device

CRFSERIALPORT

CSVRIPDLG

///

CSvrIpDlg dialog
Derived from Cdialog
Local saved file can be loaded (ip.txt) here.

CVIDEOSCREENDLG

```
///////////////////////////////////////////////////////////////////////////
 CVideoScreenDlg dialog
```

While playing media, we can see video screen and catch the status of streaming data. There two different mode in here. One is real-time playing media and the other is replaying media. As you can see the screen, there are several controls: titile, screen, slide bar and several buttons. Buttons are used when we replay medium (Play or Resume, Pause and Stop).

```
HWND GetWndScreenHandle();
void ShowVideoScreen(UINT nScreen ,UINT nMode, CString strTitle);
void EnableButtons(BOOL bStatus);
void PlayMedia(UINT nMediaType = NULL, BOOL bFlag = FALSE,
                  UINT nDurationTime = NULL);
void UpdateMediaTime(UINT nDurationTime);
```

CMSGDLG

```
///////////////////////////////////////////////////////////////////////////
 CMsgDlg dialog
```

How can user monitor the software and catch error? There is no solution to recognize error or progressing status. To display error or specific message to screen, we use this class to show error or message. Showing time is defined by timer function. After defined timer collapsed, display window will hide itself.

```
void ShowWindowView(int nCmdShow, UINT nTime = NULL);
void ShowError(UINT nErrorCode, UINT nMode);
void ShowMsg(UINT nMode, UINT nMsgCode, CString strFileName = _T(""));
void HideMsgWindow();
```

CTRACEMSGDLG

```
///////////////////////////////////////////////////////////////////////////
 CMsgDlg dialog
```

Used for showing and updating log messages.

```
void AddString(LPCTSTR lpszItem );
void ShowError(UINT nErrorCode, UINT nMode);
```

DEFINED SDK

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

Some frequently used functions are defined as SDK procedures.

[WndApi_Wnd.h]
Windows related items.

```
void WINAPI WAPI_ShutDownWnd() ;
CString WINAPI WAPI_GetItemText(CWnd *pWnd, UINT nID);
void WINAPI WAPI_FlexGridSetSellData(CMSFlexGrid *pFlexGrid, long Row, long Col,
                                      CString strData, long nColWidth,
                                      short nAlign = flexAlignCenterCenter);
```

[WndApi_Data.h]
Streaming data or logfile data related items.

```
CString WINAPI WAPI_GetModulePath();
BOOL WINAPI WAPI_DeleteFile(CString strFilename);
BOOL WINAPI WAPI_ReadWriteInitFile(CString strFilename, UINT nMode, BOOL &bRFMode,
                CString *pstrWebServerIp, CString *pstrLocalIp = NULL,
                DWORD *pWebServerIp = NULL, DWORD *pLocalIp = NULL);
BOOL WINAPI WAPI_ReadWriteAutoProcessFile(PROCESSMULTI *stData,
                 CString strFilename, UINT nMode);
void WINAPI WAPI_ReadUrlFile(URLSTRUCT *stUrl, CString strFilename);
void WINAPI WAPI_InitSet(MParam* pOldParam, MParam* pCurParam = NULL);
void WINAPI WAPI_CalculateRcvData(CONPARAM *pCONParam, MParam *pMParam,
             MParam *pOldMParam);
void WINAPI WAPI_ConvertRcvDatatoString(CONPARAM *pCONParam,
             CONPARAMSTR *pCONString);
void WINAPI WAPI_ProcessRealtimeData(CONPARAM *pCONParam,
             CONPARAMSTR *pCONString, MParam *pMParam, MParam *pOldMParam);
void WINAPI WAPI_ProcessSavedData(UINT nDts);
void WINAPI WAPI_ProcessSavedData(UINT nDts, CONPARAM *pCONParam,
             CONPARAMSTR *pCONString, MParam *pMParam);
void WINAPI WAPI_ProcessRfData(RFPARAM *pRFParam);
void WINAPI WAPI_SearchCurrentSavedDataPos(UINT nDts);
BOOL WINAPI WAPI_SveRcvData(CONPARAM *pCONParam);
BOOL WINAPI WAPI_ReadSveData(CONPARAM *pCONParam);
void WINAPI WAPI_FinishPlaying(UINT nMode);

void WINAPI WAPI_ReadSvedData();
void WINAPI WAPI_DeleteUploadedData();
BOOL WINAPI WAPI_ReadGatewayFile(PMIB_IPFORWARDROW pArrow);
BOOL WINAPI WAPI_WriteGatewayFile(PMIB_IPFORWARDROW pArrow);
```

[WndApi_Chart.h]
Chart related items.

```
void WINAPI WAPI_InitChartData(CWnd *pWnd, DOUBLE nCur,
             UINT ID_CUR, double *pdCur, UINT ID_MAX, double *pdMax,
             UINT ID_MIN, double *pdMin, UINT ID_AVE, double *pdAve);
void WINAPI WAPI_InitChartData(CWnd *pWnd, FLOAT nCur,
             UINT ID_CUR, FLOAT *pdCur, UINT ID_MAX, FLOAT *pdMax,
             UINT ID_MIN, FLOAT *pdMin, UINT ID_AVE, FLOAT *pdAve);
void WINAPI WAPI_InitChartData(CWnd *pWnd, UINT nCur,
             UINT ID_CUR, UINT *pdCur, UINT ID_MAX, UINT *pdMax,
             UINT ID_MIN, UINT *pdMin, UINT ID_AVE, UINT *pdAve);
void WINAPI WAPI_InitChartData(CWnd *pWnd, INT nCur,
             UINT ID_CUR, INT *pdCur, UINT ID_MAX, INT *pdMax,
             UINT ID_MIN, INT *pdMin, UINT ID_AVE, INT *pdAve);
void WINAPI WAPI_UpdateChartData(CWnd *pWnd, UINT nTotal, DOUBLE nCur,
             UINT ID_CUR, double *pdCur, UINT ID_MAX, double *pdMax,
             UINT ID_MIN, double *pdMin, UINT ID_AVE, double *pdAve);
void WINAPI WAPI_UpdateChartData(CWnd *pWnd, UINT nTotal, FLOAT nCur,
             UINT ID_CUR, FLOAT *pdCur, UINT ID_MAX, FLOAT *pdMax,
             UINT ID_MIN, FLOAT *pdMin, UINT ID_AVE, FLOAT *pdAve);
void WINAPI WAPI_UpdateChartData(CWnd *pWnd, UINT nTotal, UINT nCur,
             UINT ID_CUR, UINT *pdCur, UINT ID_MAX, UINT *pdMax,
             UINT ID_MIN, UINT *pdMin, UINT ID_AVE, UINT *pdAve);
void WINAPI WAPI_UpdateChartData(CWnd *pWnd, INT nTotal, INT nCur,
```

```
UINT ID_CUR, INT *pdCur, UINT ID_MAX, INT *pdMax,
UINT ID_MIN, INT *pdMin, UINT ID_AVE, INT *pdAve);
```

Listed below components are derived from MFC original classes. these components are just do changing form color or UI design. Some useful examples can be found on the Internet (http://www.codeguru.com).

CSPLASHWND

///
CSplashWnd window
Derived from CWnd

CPICTURE

///
CPicture wrapper class
Derived from COleDispatchDriver

CSTATICLABEL

///
CStaticLabel window
Derived from CStatic

CBUFFERSTRUCT

CXPSTYLEBUTTONST

Derived from CButtonST

CTEXTROTATOR

///
class CTextRotator

PROGRESSBAR

///
ProgressBar window
Derived from CProgressCtrl

CTOOLBAREX

Derived from CToolBar

CEXCHECKBOX

///
CExCheckBox window
Derived from CButton

CGRADIENTSTATIC

Derived from CStatic

CLOGFONT

CITEMBITMAP

Class CItemBitmap

CEDITMASK

///
CEditLabel window

CCONTROLTOOLTIP

///
CControlToolTip window
Derived from CWnd

CCOLORLISTBOX

///
CColorListBox window
Derived from CListBox

COLEFONT

///
COleFont wrapper class
Derived from COleDispatchDriver

CRASCONNECTIONDLG

Derived from CDialog

CCOLORFORMVIEW

Derived from CFormView

CARRAY<CTABITEM*,CTABITEM*>

CEXSTATIC

///
CExStatic window
Derived from CStatic

CARRAY<CEXPROPERTYPAGE*,CEXPROPERTYPAGE*>

CROWCURSOR

///
CRowCursor wrapper class
Derived from COleDispatchDriver

CMSFLEXGRID

///
CMSFlexGrid wrapper class
Derived from CWnd

CEXPROPERTYSHEET

Derived from CEXTabCtrl

CEDITLABEL

Derived from CEdit

CBUTTONST

Derived from CButton

CCONTROLPOS

CAUTOPROCESSDLG

///
CAutoProcessDlg dialog

Derived from CDialog

CEXTOOLBARWND

//
CCustomizeDialog dialog
Derived from CWnd

AFX_DLLVERSIONINFO

CEXPROPERTYPAGE

XPropertyPage.h : header file
Derived from CDialog

CBITMAPMENU

Derived from CMenu

CTHEMEHELPERST

CXPBUTTON

//
CXPButton window
Derived from CButton

CDOCKBAREX

CDockBarEx Class

CEXTABCTRL

Derived from CWnd

APPENDIX A. DEFINED PACKET

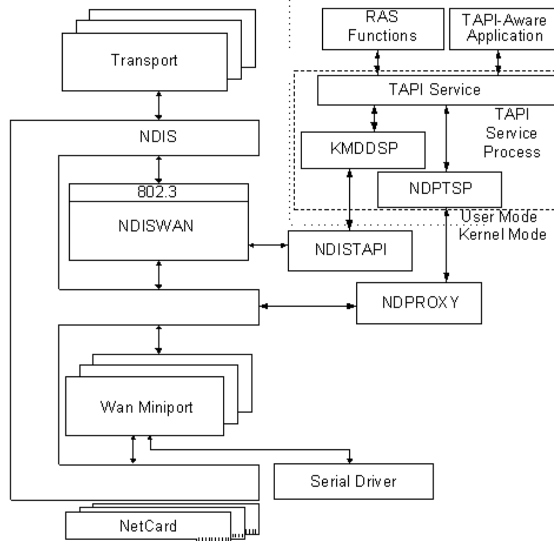
Used Protocol is HTTP(Hyper Text Transfer Protocol). There are two kinds of method to send data using HTTP such as POST and GET method. We here used GET method. A data packet consists of several arguments. Each argument is divided by separator (minus character "-"). All data are converted to character type before send. If a value in a data packet is not exists, we just send character "F".
Error Code :
0 : Finished Media
100 : File open error
101 : File processing error
150 : Cannot Connect to Media Server
200 : Streaming server connecting error
201 : Streaming data processing error
202 : Streaming data does not comes out (Received at least one packet)
203 : Streaming data does not comes out
500 : RAS connecting error
501 : Reset Mobile (cannot make a call)
502 : Reset Mobile (RF data does not come out)
900 : Internal error

Data Packet Arguments :
1 : Service Type(MPEG2:1, H.26L:2, Wavelet:3)
2 : Network ID (1:EVDO, 2:1x)
3 : Measure Success/Fail (1:Success, 2:Fail)
4 : Fail Reason (Error Code)
5 : Measure Start Time(MMDDHHMMSS)
6 : Media Content ID
7 : Media Session ID
8 : Measure Duration Time(MMSS)
9 : Signaling Time(ms)
10 : Initial Buffering Time (ms)
11 : Average Video Packet Loss Rate(%)
12 : Average Video Rate (Byte Per Second)
13 : Average Video Frame Delivery Time(ms)
14 : Average Video Quality Indicator(0-1)
15 : Average Video Delay Jitter
16 : Average Video Packet Throughput(%)
17 : Average Video Frame Size(byte)
18 : Average Video Frame Error Rate(%)
19 : Average Video Packet Header Size (byte)
20 : Average Video Frame Rate(fps)
21 : Average Video Buffer Status(EA)
22 : Average Video Retransmission Ratio(%)
23: Average Audio Packet Loss Rate(%)
24 : Average Audio Rate (Byte Per Second)
25 : Average Audio Frame Delivery Time(ms)
26 : Average Audio Quality Indicator(0-1)
27 : Average Audio Delay Jitter
28 : Average Audio Packet Throughput(%)
29 : Average Audio Frame Size(byte)
30 : Average Audio Frame Error Rate(%)
31 : Average Audio Packet Header Size (byte)
32 : Average Audio Frame Rate(fps)
33 : Average Audio Buffer Status(EA)

34 : Average Audio Retransmission Ratio(%)
35 : The most frequently measured Pilot No
36 : The most frequently measured Channel No
37 : Average Rx Power
38 : Average Ec/Io

APPENDIX B. RAS ARCHITECTURE

This shows the relationship between Remote Access Service (RAS) and how drivers for wide area network cards such as ISDN, X.25, and Switched 56 adapters use the services of NDIS and NDISWAN to communicate in both standard WAN and connection-oriented WAN environments.
The following figure shows the RAS architecture.



RAS Architecture

The following describes the NDISWAN intermediate NDIS driver, the NDISTAPI driver, and the NDPROXY driver and provides a more detailed view of the entire RAS system.
The components of WAN, RAS, and TAPI that are shown in the preceding figures are described next.

RAS Functions

The RAS set of functions allows user-mode applications to make RAS connections. After a RAS connection is established, applications can connect to network services using standard network interfaces such as Windows Sockets, NetBIOS, Named Pipes, or RPC.

Transports

The RAS system component provides transports such as PPP Authentication (PAP, CHAP) and network configuration protocols (IPCP, IPXCP, NBFCP, LCP, and so forth). A WAN miniport driver implements only PPP media-specific framing.

TAPI Service

The TAPI service (*tapisrv.exe*) presents the Telephony Service Provider Interface (TSPI) of different service providers to TAPI-aware applications. Applications use specific service providers to communicate with specific device types. These service providers are DLLs that run in the context of the TAPI service process. The operating system supplies service providers that both standard and CoNDIS WAN miniport drivers can use to communicate with user-mode applications.

KMDDSP

This component is a service provider DLL that runs in the context of the TAPI service process. The *kmddsp.tsp* component presents a TSPI interface to TAPI-aware applications so that NDISTAPI can communication with user-mode applications. This component converts user-mode requests to corresponding TAPI OIDs for NDISTAPI.

NDISTAPI

This component implements TAPI with a kernel-mode portion of the TAPI interface. The *ndistapi.sys* component communicates with standard WAN miniport drivers by routing TAPI-related OID requests with the **NdisRequest** function to the appropriate standard WAN miniport driver.

NDPTSP

This component is a service provider DLL that runs in the context of the TAPI service process. The *ndptsp.tsp* component presents a TSPI interface to TAPI-aware applications so that NDPROXY can

communication with user-mode applications. This component converts user-mode requests to corresponding TAPI-CO-related OIDs for NDPROXY.
NDPROXY
This component implements TAPI by encapsulating TAPI parameters in NDIS structures when making and accepting calls. The *ndproxy.sys* component communicates with TAPI through the TSPI interface of NDPTSP. The *ndproxy.sys* component communicates through NDIS with NDISWAN and a CoNDIS WAN NIC miniport driver. This component presents a client interface to a miniport driver and a call manager interface to NDISWAN. NDISWAN presents a client interface to this component. A miniport driver presents a call manager interface to this component. This component enumerates TAPI capability of a CoNDIS WAN miniport driver by calling the **NdisCoRequest** function with TAPI-CO-related OIDs. This component also registers the TAPI-specific address family, creates virtual connections (VCs), makes and accepts calls, and activates VCs so that data can be sent and received on those VCs.
NDISWAN
The NDISWAN intermediate NDIS driver supports PPP protocol/link framing, compression, and encryption. NDISWAN supports both standard and connection-oriented miniport drivers. The *ndiswan.sys* driver communicates with standard WAN NIC drivers through two interfaces:
NDIS WAN interface
NDIS miniport driver interface
The *ndiswan.sys* driver communicates with CoNDIS WAN miniport drivers through the connection-oriented miniport driver interface.
Serial Driver
This component is a standard device driver for internal serial ports or multiport serial cards. The built-in asynchronous WAN miniport driver for Windows® 2000 and later uses the internal serial driver for modem communications. Any driver that exports the same functions as the serial driver will work with the built-in asynchronous WAN miniport driver.
X.25 vendors can choose to implement serial driver emulators for the X.25 card. In this case, each virtual circuit on the X.25 card appears as a serial port (with an X.25 PAD attached to it). The connection interface must correctly emulate serial signals such as DTR, DCD, CTS, RTS, and DSR.
X.25 vendors who choose to implement a serial driver emulator for their X.25 card must also make an entry for their PAD in the *pad.inf* file. This file contains the command/response script needed to make a connection through the X.25 PAD. For more information about the *pad.inf* file, see the Remote Access Service in the Platform SDK.
WAN Miniport Driver
Depending on environment, ISDN, Switched 56, and X.25 vendors should write either a standard or CoNDIS WAN miniport driver for their NICs.

Built on Thursday, July 19, 2001 exerted from MSDN

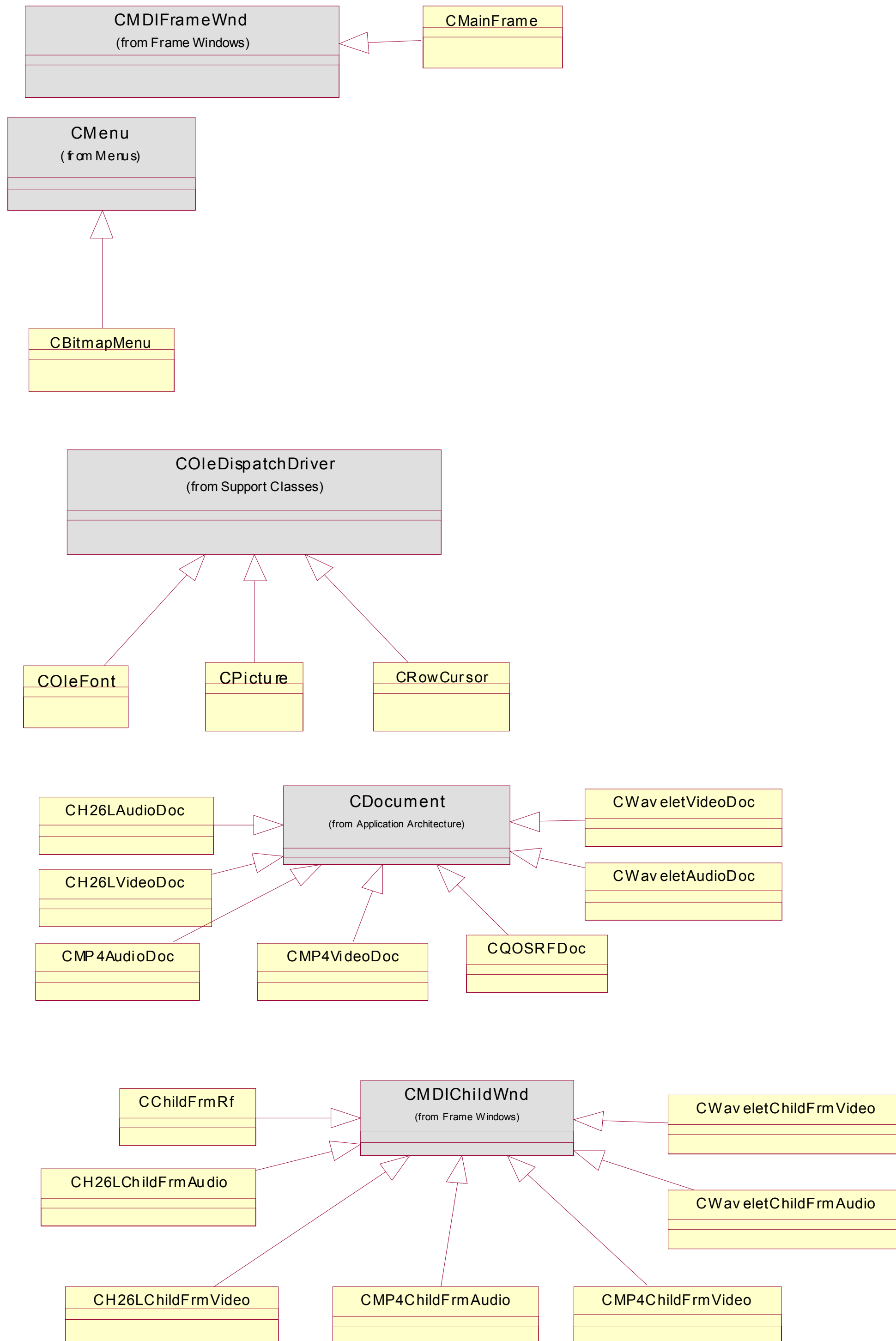
APPENDIX C. DATA STRUCTURE

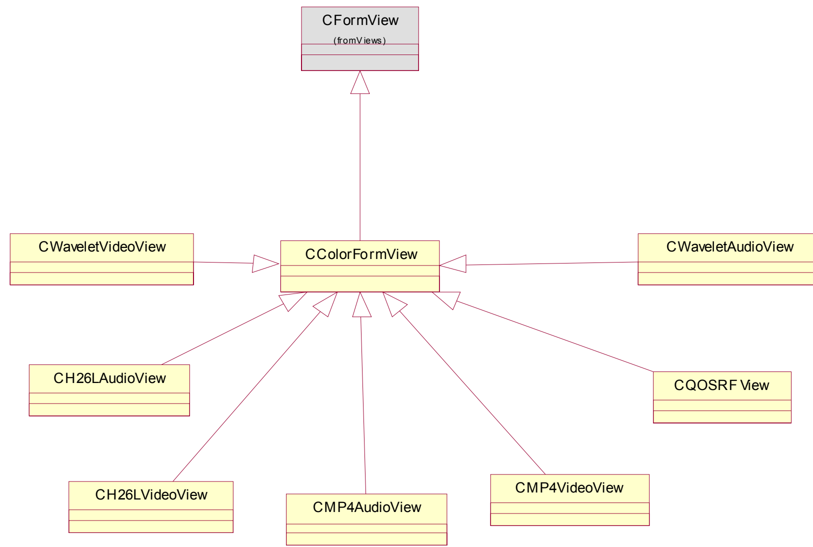
Original streaming data (structure name is Mparameter) must be calculated to use in our UI. Here CONPARAM structure shown below are used to save temporary. Some calculate formulas are commented right side of value name.

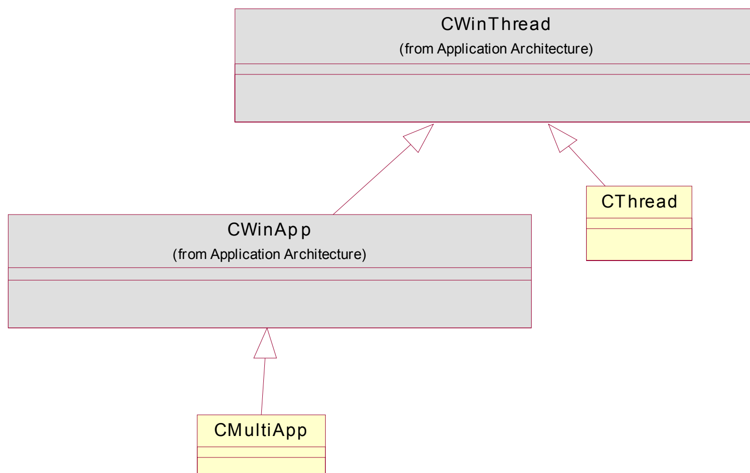
```
untimeLastDecode_A;          // DTS(timestamp)
untimeLastDecode_V;

// 1. Signaling Time
unSignalingTime_A;
unSignalingTime_V;           // = timeSignalDESC + timeSignalSETUP +
                             timeSignalPLAYwait
// 2. Packet Loss Rate
dPacketLossRate_V;
dPacketLossRate_A;           // = ( numLostPackets /
                             (numRcvPackets + numLostPackets + numErrorPackets))
unRcvPackets_A;              // Receive Packets
unRcvPackets_V;
unLostPackets_A;             // Lost Packets
unLostPackets_V;
unErrorPackets_A;            // Error Packets
unErrorPackets_V;

// 3. Bit Rate
dCurrentBitRate_A;
dCurrentBitRate_V;           // = byteReceived_v / timeLastDecode_v
unReceivedByte_A;
unReceivedByte_V;

// 4. Frame Delivery Time(Buffering Time)
unFrameDeliveryTime_A;
unFrameDeliveryTime_V;

// 5. QualityIndicator
fQualityIndicator_A;
fQualityIndicator_V;

// 6. CurrentJitter
unDelayJitter_A;
unDelayJitter_V;

// 7. Throughput
dThroughput_A;
dThroughput_V;               // = ( numRcvPackets - numErrorPackets ) /
                             ( numRcvPackets + numLostPackets)
// 8. CurrentMaxFrameSize
unFrameSize_A;
unFrameSize_V;

// 9. Frame Error Rate
dFrameErrorRate_A;
dFrameErrorRate_V;           // = numErrorFrame / (numRcvFrame + numLostFrame + numErrorFrame)
unRcvFrame_A;
unRcvFrame_V;
unLostFrame_A;
unLostFrame_V;
unErrorFrame_A;
unErrorFrame_V;
```

```
// 10. Retransmission Ratio
unRetRequest_A;
unRetRequest_V;
unRetSuccess_A;
unRetSuccess_V;
dRetransmissionRatio_A;     // = numRetRequest_a / numRetSuccess_a * 100.
dRetransmissionRatio_V;     // = numRetRequest_v / numRetSuccess_v * 100.

// 11. Initial Buffering Time
unInitBufferingTime_A;
unInitBufferingTime_V;

// 12. Header Size
unPacketHeaderSize_A;
unPacketHeaderSize_V;

// 13. Original Frame Rate
fOriginFrame_A;
fOriginFrame_V;
fFrameRate_A;               // Displayed video frame per second
                            = unDecodedAU_A / untimeLastDecode_A * 1000 (sec)
fFrameRate_V;               // Displayed Audio frame per second
                            = unDecodedAU_V / untimeLastDecode_V * 1000 (sec)

unBufferStatus_A;           // Buffer status
unBufferStatus_V;
```

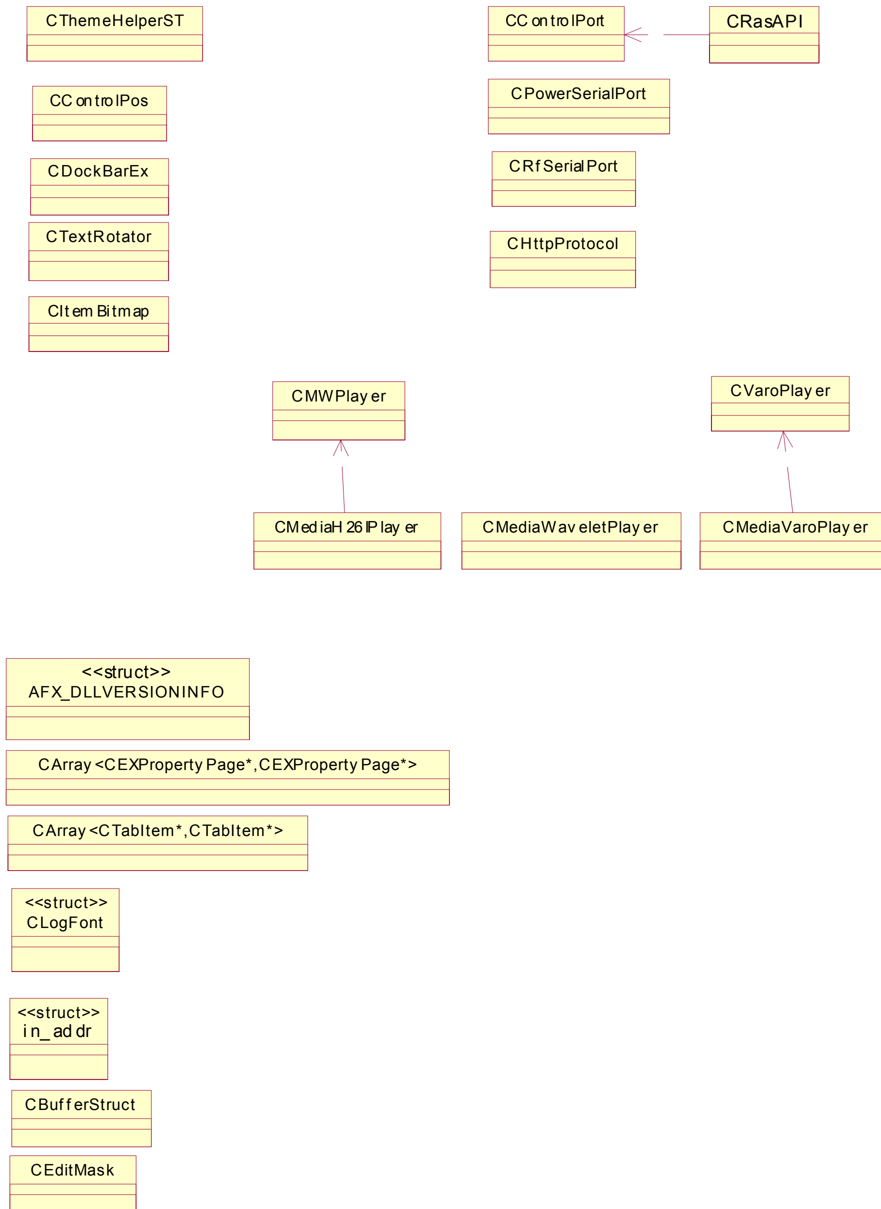
APPENDIX D. DATA FLOW DIAGRAM




CFormView
(fromViews)
CWaveletVideoView
CColorFormView
CWaveletAudioView
CH26LAudioView
CQOSRF View
CH26LVideoView
CMP4AudioView
CMP4VideoView

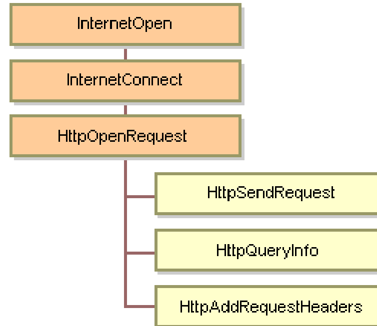

CWinThread
(from Application Architecture)
CThread
CWinApp
(from Application Architecture)
CMultiApp

CStatusAudio
CStatusVideo
CStatusRf
CEXPropertyPage
CRasConnectionDlg
CSaveOnceDlg
CStatusDlg
CSaveContDlg
CVideoScreenDlg
CMSFlexGrid
CMsgDlg
CSplashWnd
CSvrIpDlg
CDialog
(from Dialog Boxes)
CAboutDlg
CControlToolTip
CExToolbarWnd
CEXPropertySheet
CEXTabCtrl
CWnd
(from Window Support)
CListBox
(from Controls)
CColorListBox
CEditLabel
CEdit
(from Controls)
CControlBar
(from Control Bars)
CButton
(from Controls)
CStatic
(from Controls)
CStatusDlgBar
CExCheckBox
CToolBar
(from Control Bars)
CXPButton
CExStatic
CStaticLabel
CToolBarEx
CButtonST
CXPStyleButtonST
CGradientStatic


CThemeHelperST
CC on tro lPort
CRasAPI
CC on tro lPos
CPowerSerialPort
CDockBarEx
CRf SerialPort
CTextRotator
CHttpProtocol
CIt em Bitmap
CMWPlay er
CVaroPlay er
CMed iaH 26 lP lay er
CMediaWav eletPlay er
CMediaVaroPlay er
<<struct>>
AFX_DLLVERSIONINFO
CArray <CEXProperty Page*,CEXProperty Page*>
CArray <CTabItem*,CTabItem*>
<<struct>>
CLogFont
<<struct>>
i n_ ad dr
CBuff erStruct
CEditMask

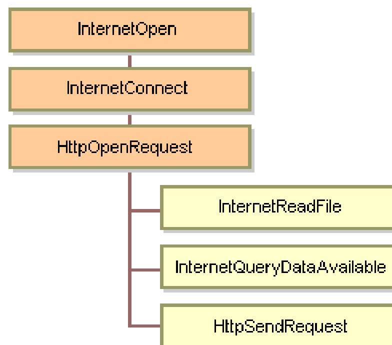
APPENDIX E. ACCESSING THE HTTP PROTOCOL

Use the HTTP functions provided by WinInet to use the HTTP protocol to access resources on the Internet. The following illustration shows the relationships of the WinInet functions used to access the HTTP protocol. Shaded boxes represent functions that return **HINTERNET** handles, while the plain boxes represent functions that use the **HINTERNET** handle created by the function on which they depend.



HttpAddRequestHeaders, **HttpQueryInfo**, and **HttpSendRequest**, are dependent on the **HINTERNET** handle created by **HttpOpenRequest**.

The following illustration shows the WinInet functions that use the **HINTERNET** handle created by **HttpOpenRequest** after it is sent by **HttpSendRequest**. The shaded boxes represent functions that return **HINTERNET** handles, while the plain boxes represent functions that use the **HINTERNET** handle created by the function on which they depend.



After **HttpSendRequest** has been used on the handle returned by **HttpOpenRequest**, **InternetQueryDataAvailable**, and **InternetReadFile**, can be used on that handle.

To use the HTTP WinInet functions

Call the InternetOpen function to initialize an Internet handle.

InternetOpen creates the root **HINTERNET** handle used to establish the HTTP session. The **HINTERNET** is used by all subsequent functions.

Call InternetConnect using the **HINTERNET** returned by **InternetOpen** to create an HTTP session.

When calling **InternetConnect**, specify INTERNET_DEFAULT_HTTP for the *nServerPort* parameter and INTERNET_SERVICE_HTTP for the *dwService* parameter.

InternetConnect uses the handle returned by **InternetOpen** to create a specific HTTP session. **InternetConnect** initializes an HTTP session for the specified site, using the arguments passed to it and creates **HINTERNET** that is a branch off the root handle. **InternetConnect** does not attempt to access or establish a connection to the specified site.

Call HttpOpenRequest to open an HTTP request handle.

HttpOpenRequest uses the handle created by **InternetConnect** to establish a connection to the specified site.

Call HttpSendRequest using the handle created by the **HttpOpenRequest** to send an HTTP request to the HTTP server.

Call InternetReadFile to download data.

–Or–
Call InternetQueryDataAvailable to query how much data is available to be read by a subsequent call to **InternetReadFile**.
Call InternetCloseHandle to close the handle created by **HttpOpenRequest**.
Call **InternetCloseHandle** to close the HTTP session created by **InternetConnect**.
Call **InternetCloseHandle** to close the handle created by **InternetOpen**.

APPENDIX F. POWER CONTROLLER & IPC

To check the system is operating or not, power controller and the software system must communicate each other. Here is the message to communicate.

T0 : Initialize controller	Response : R0
T1 : ACK to indicate software is acting correctly	Response : R1
T2 : Reset Mobile	Response : R2
T3 : System reset	Response : R3
T4 : IPC turned off	Response : R4*
T5 : System shutdown	Response : R5

As you can see above list, prefix T- indicate newly occurred message and prefix R- is response message. All messages (T-) get through to the power controller except T4. T4 is occurred in power controller when the administrator turn off the system and send it to the system software indicating power controller will shutdown OS system in 30 seconds.

*) This message does not comes out from the power controller. It must be send message R4 to power controller by the system software.