# Implementing Tuner Arbitration
## Application Note

DPCIAN1A
September 7, 1999

CONEXANT™

**Reader Response:** Conexant strives to produce quality documentation, and welcomes your feedback. Please send comments and suggestions to conexant.tech.pubs@conexant.com. For technical questions, contact your local Conexant sales office or field applications engineer.

# Implementing Tuner Arbitration

## Introduction

This application note introduces a new TV tuner arbitration scheme that can be used with the Fusion video for Windows drivers.

Currently, if an application is using the TV tuner, any attempt to start another video for Windows application will fail. You get a dialog box indicating an error with the video capture driver. The error message does not indicate that this error occurs because there is only one video capture device in the system. Because it is not possible to have two instances of the same driver running simultaneously, the driver sends an error message and the video for Windows application must terminate.

This problem becomes worse when you install software that supports data casting. A typical application is the WaveTop version 2.0 developed and distributed by Wavo Corporation. The WaveTop application decodes the VBI data broadcast over the local PBS channel and stores the decoded data on the hard drive. To gain access to the VBI data, the WaveTop application must use the TV tuner while the PC is running. Therefore, you will get an error message if you invoke a TV viewer application while the WaveTop application is running. You must stop the WaveTop application manually before invoking a TV viewer application.

The TV tuner arbitration scheme is invented by Wavo Corporation to overcome the above-described problem. If implemented by all video for Windows applications, this scheme allows applications to share the TV tuner and provides user-friendly environments for users of TV capture cards.

## *Implementing the TV Tuner Arbitration Class in a Video for Windows Application*

To illustrate the process of implementing the tuner arbitration class in a video for Windows application, our sample application, ExtremeTV, is included as an example. Start with the application before implementing the tuner arbitration class, and work your way to a fully tuner-arbitration-compatible video for Windows application.

### *Downloading ExtremeTV*

Conexant provides the ExtremeTV application, including source code, on our FTP site. To obtain a copy of the full source, follow the instructions below:

1. Run your web browser and supply this address:
   ftp://fusion:broadcast@www7.conexant.com/fusion/VXD/Applications/ExTV/ExTV.zip
2. When prompted by the browser, save the file in a convenient folder.
3. Unzip the archive ExTV.zip to get the source code and the ExtremeTV installer.
4. Note that the TV.mdp workspace is created with Visual C++ 4.0 and the application must be built with Visual C++ 4.0.

### *Incorporating the Tuner Arbitration Class*

1. To automatically launch the Visual C++ 4.0 application, double-click the TV.mdp project workspace file.
2. Add the header file tunerarbitration.h and source file tunerarbitration.cpp into the project.
3. Use the build option to make the ExtremeTV application.
4. Make sure the class wizard adds the CTunerArbitration class after successful compilation.
5. If the application is written in C, you first convert the C source file into a CPP source file before attempting to incorporate the tuner class. In the case of ExtremeTV, you must first convert pbtv.c to pbtv.cpp. You may also need to include the extern “C” syntax in the other C files to ensure that the compiler will successfully build the application. Details on using the extern “C” syntax are in most programming guides.
6. Include the header file tunerabitration.h in pbtv.cpp and resolve any compile issues.
7. Once the header file is included, declare an instance of the CTunerArbitration object.
8. Add the following line in the pbtv.cpp file: CTunerArbitration cTuner;

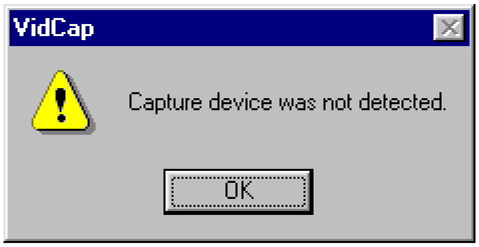
## *Why Not Simply Use the FindWindow Function?*

It is possible to prevent a video for Windows application from running twice and thus avoid the problem of Tuner arbitration by using the FindWindow function. The following code fragment is used in the WinMain function of ExtremeTV to accomplish this task:

```
HWND FirsthWnd;
    if(FirsthWnd=FindWindow(TV_APP_CLASS, TV_APP_WIN))
    {
       SetForegroundWindow(FirsthWnd);         // bring main window to top
       ShowWindow(FirsthWnd, SW_RESTORE);      // restore app in case it is
                                                 minimized

       return(FALSE);                          // do not run second instance
    }
```

This approach is very effective and should be used in all video for Windows applications. The problem is that it does not solve the Tuner arbitration problem completely. Normally, a video for Windows application ignores the fact that a different video for Windows application may be using the capture driver and connects to it using capDriverConnect function call. If ExtremeTV is running, invoking VidCap32 generates the following error message:



Capture device was not detected.

The challenge is to make sure that the application does not try to load the video capture driver when another application is using either the capture driver or the TV tuner. This can be accomplished using the Tuner Arbitration scheme described in this application note.

## *Adding Tuner Arbitration Support*

### Error Prevention

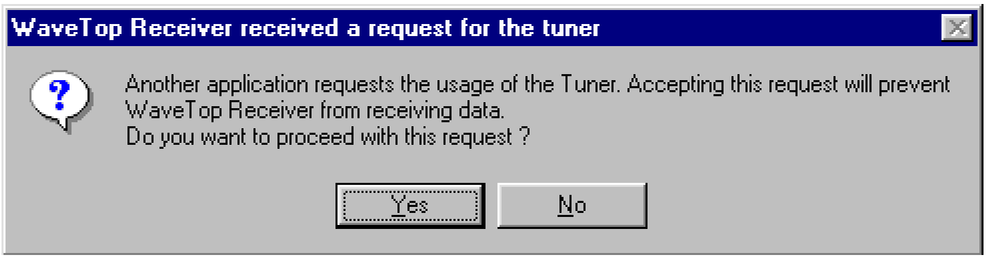
1. To prevent the “error” from occurring, check to see if another application is using the TV tuner before attempting to connect to the capture driver. This service is available via the CTunerArbitration class. The CTunerArbitration class registers the appropriate Windows message and creates a thread to acquire the TV tuner at regular intervals if it is not already available.
2. Use the member methods AcquireTunerNow or AcquireTunerWhenAvailable to check the availability of the TV tuner. If the flag bTunerAcquired is set (i.e., the TV tuner has been acquired by our application), the two methods will return TRUE.

3. You can check the tuner availability in the InitInstance function call. The method CTunerArbitration::TunerOwned checks if the Tuner is owned. Add the following:

```
if(cTuner.TunerOwned())
{
        if(!InitVideo(hInstance, nCmdShow))
                return (FALSE);
}
```

### What Happens When You Cannot Acquire the TV Tuner?

If we cannot acquire the tuner, do we simply terminate the application? That would mean that if WaveTop happens to be running all the time, your application would never get a chance to run. Fortunately, the WaveTop application is listening in on all TV tuner requests whenever it is running. When our application tries to acquire the TV tuner, the WaveTop application displays the following dialog box to prompt the user’s response:



WaveTop Receiver received a request for the tuner

Another application requests the usage of the Tuner. Accepting this request will prevent WaveTop Receiver from receiving data.
Do you want to proceed with this request ?

- If the user responds with a “Yes”, the WaveTop application releases the TV tuner. Once the TV tuner is released, the thread started by our CTunerArbitration object (CTuner) can acquire the TV tuner.
- When the CTuner object acquires the tuner, the application can then create the video window and connect to the capture driver using the capCreateCaptureWindow and capDriverConnect calls.

### How Do You Know You Have Acquired the TV Tuner?

The CTunerArbitration object takes care of it in the method CTunerArbitration::ProcessMessage. What you must do is make sure that the tuner arbitration messages are processed in the main window of the ExtremeTV application.

1. Add this to the necessary code fragment in the MainWndProc callback function:

```
int iRet;

iRet=cTuner.ProcessMessage(hWnd, wParam, lParam);

if(iRet==ACQUIRED_TUNER)
        InitVideo(hInst,1);
```

2. Adding the step one code fragment requires us to create the main window even if we are unable to acquire the TV tuner right away.
3. You must also create a new function that will create the video window when it is appropriate to do so. The InitVideo function creates the video window and connects to the capture driver. When the Extreme TV application closes, the CTunerArbitration class terminates the thread that it started and frees up the tuner for other applications.

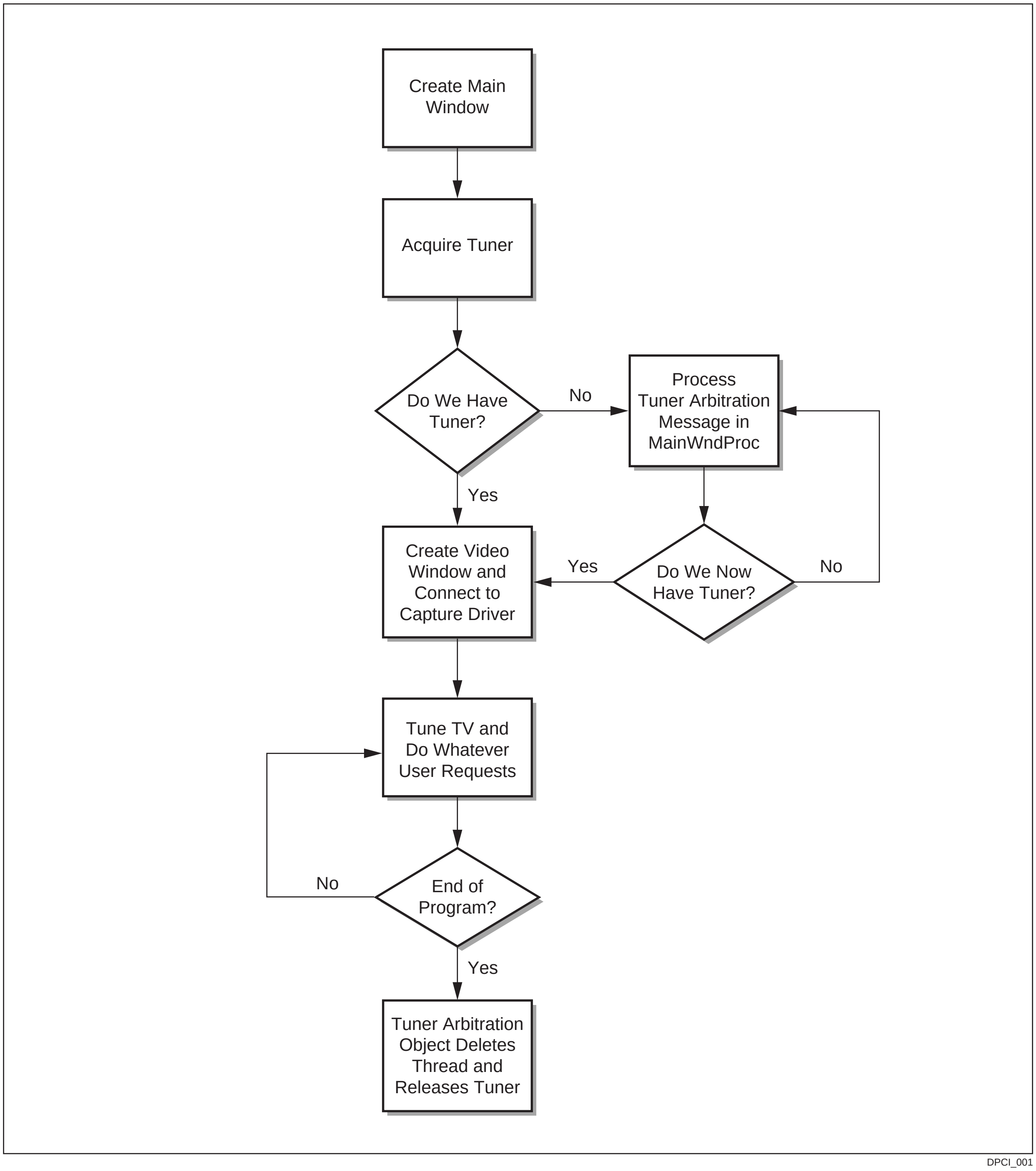
If the WaveTop application was running in the background before ExtremeTV was launched, it would get a chance to acquire the TV tuner and continue its data collection tasks. This makes it easy to use the WaveTop application as you do not have to worry about stopping the WaveTop application before watchingTV. Similarly, you do not have to restart the WaveTop application to re-enable the data collection process.

## *Example ExtremeTV with Tuner Arbitration Support*

The complete source to the new ExtremeTV program is available on the following Conexant ftp site: ftp://fusion:broadcast@www7.conexant.com/fusion/VXD/Applications/

The flowchart included with this application note should also help the programmer understand how to implement the Tuner Arbitration class. Please note that once the main window is created, the callback function MainWndProc will continue to process tuner arbitration messages until the application is closed.

***Figure 1. Flow Chart of Tuner Arbitration Control***


Create Main Window
Acquire Tuner
Do We Have Tuner?
No
Process Tuner Arbitration Message in MainWndProc
Yes
Create Video Window and Connect to Capture Driver
Yes
Do We Now Have Tuner?
No
Tune TV and Do Whatever User Requests
No
End of Program?
Yes
Tuner Arbitration Object Deletes Thread and Releases Tuner
DPCI_001

## *Complete Tuner Arbitration Source*

### *Header file tunerarbitration.h*

```
class CTunerArbitration {
public:
      CTunerArbitration();
      ~CTunerArbitration();

      // verify if the Named-Mutex (the tuner) is owned or not
BOOL  TunerOwned(void);

      // processing of the tuner arbitration specific Windows message
int   ProcessMessage(HWND hWnd,WPARAM wParam,LPARAM lParam);

      // returns the Win msg registered specifically for the T.A.
UINT  TunerArbitrationSpecificWinMsg(void){return m_WinMsg;}

      // if the tuner is not available, this object will wait for a
      TUNER_AVAILABLE msg
      // and then try to acquire it.
BOOL  AcquireTunerWhenAvailable(void);

      // attempt to immediatly acquire the tuner. if not available, a
      TUNER_REQUEST msg will be
      // broadcasted.  The response (a win msg) must be processed by
      ProcessMessage().
BOOL  AcquireTunerNow(void);

      // thread handling user feedback
void  GetUserFeedbackThread(void);

      // thread polling for tuner when interacting with apps that do not comply
      with TA mechanism
void  TunerAvailabilityThread(void);

private:
      // acquire the Named-Mutex before calling InitSurfVideo() (surfvapi.dll)
BOOL  AcquireTuner(void);
      // release the Named-Mutex before calling InitSurfVideo() (surfvapi.dll)
BOOL  ReleaseTuner(void);

BOOL  BroadcastMessage(LPARAM lParam);
BOOL  SendMessage(HWND hDestWnd, LPARAM lParam);

BOOL  AcquireIntercastVideoStack();
BOOL  ReleaseIntercastVideoStack();
void  SaveTunerReleaseConfirmationFromUser();
void  ReadTunerReleaseConfirmationFromUser();

BOOL  CreateTunerAvailabilityThread(void);
void  DestroyTunerAvailabilityThread(void);

BOOL  PostMsgToProcessMessageWnd(LPARAM lParam,WPARAM wParam);

      // Win msg registered specifically for the T.A.
UINT  m_WinMsg;

      // mutex
HANDLE      m_hTAMutex,
            m_hTunerAvThread,
            m_hUserFeedbackThread,
            m_hEvStopCheckingTunerAv;

      // indicate if this object owns the tuner or not
BOOL  m_bOwnMutex;

      // indicate if this object owns the Intercast Video Stack or not (same as
      above)
BOOL  m_OwnVideoStack;

      // tuner request policy:
int   m_RequestPolicy;
int   m_RequestState;

      // avoid re-acquiring the tuner we just released.
BOOL  m_bSkipTheNextAttemptToAcquireTuner;

HWND  m_hICWnd,
      m_hMessageWnd;

BOOL  m_bTunerReleaseConfirmationFromUser;

BOOL  m_bPostedCmdStatus;
HANDLE            m_hEvWaitForProcessedMsg;
CRITICAL_SECTION  m_csPostMsgToProcessMessageWnd;

BOOL  m_bReadyToProcessARequest;

      // temporary storage used when calling user feedback thread
WPARAM      m_wParam;

};

// possible return values for ProcessMessage()
#define NOTHING               0
#define ACQUIRED_TUNER  1
#define LOST_TUNER      2
```

### Source file tunerarbitration.cpp

```
#include <windows.h>
#include "TunerArbitration.h"


static void GetUserFeedback_static(PVOID pParam);

// ----------------------------------------------------------------------
//
//    Tuner Arbitration : definitions common among participants apps.
//
// ----------------------------------------------------------------------

// Use this for the call to CreateMutex.  By using a named mutex
// we'll be guaranteed of grabbing the same mutex for TV tuner
// arbitration.
#define MUTEX_NAME                  "IC Tuner Arbitration Mutex"

// Use this for the call to RegisterWindowMessage.  By using the
// same string across all applications we guarantee that we'll get
// the same message ID for everyone.
#define WINDOWS_MESSAGE_NAME        "IC Tuner Arbitration Message"

// The following subtypes and subtype parameters are used in the
// lParam of BroadcastSystem message.  lParam is a 32-bit parameter,
// so we form it by taking the bitwise OR of the subtype shifted left
// by 16 bits with the subtype parameter.  E.g.,
//
//                      (IC_TUNER_REQUEST<<16) | TUNER_REQUESTED
//
// This forms a message to request the tuner.
//
// N.B.  You should only use the IC_TUNER_RESPONSE parameters with
// that subtype, and similarly for IC_TUNER_REQUEST.

// Message formation macros
//#define MKTUNERMSG(s, sp) ((unsigned)((s<<16) | sp))
#define MKTUNERMSG(s, sp) ((LPARAM)((s<<16) | sp))

// Subtypes
#define IC_TUNER_REQUEST 0x8000
#define IC_TUNER_RESPONSE 0x0001

// Subtype parameters

// Parameters for IC_TUNER_REQUEST
#define TUNER_REQUESTED                   0x0001
#define TUNER_AVAILABLE                   0x0002
```

```
// Parameters for IC_TUNER_RESPONSE
#define TUNER_REQUEST_ACCEPTED            0x7fff
#define TUNER_REQUEST_WAIT_FOR_USER       0x7ffe
#define TUNER_REQUEST_DENIED              0xdead

// Private Window Messages lParam values
#define PRIVATE_IC_TUNER_ACQUIRE0xffffffff
#define PRIVATE_IC_TUNER_RELEASE0x11111111

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

// Policy for tuner requests
#define ASK_USER             1
#define NEVER_RELEASE_TUNER  2
#define ALWAYS_RELEASE_TUNER 3


// Request State
#define STATE_REQ_IDLE                          0
#define STATE_REQ_WAITING_FOR_RESPONSE          1
#define STATE_REQ_WAITING_FOR_TUNER_AVAILABLE   2

//---------------------------------------------------------------------------
//    public methods
//---------------------------------------------------------------------------
CTunerArbitration::CTunerArbitration()
{
      m_RequestState=         STATE_REQ_IDLE;
      m_OwnVideoStack=        FALSE;

      //why ask the user when we're running a TV app?? We should simply deny
      all requests
    //m_RequestPolicy=        ASK_USER;
      m_RequestPolicy=        NEVER_RELEASE_TUNER;
      m_bOwnMutex=            FALSE;
      m_hTunerAvThread=       NULL;
      m_hUserFeedbackThread=  NULL;
      m_bSkipTheNextAttemptToAcquireTuner=FALSE;
      m_bPostedCmdStatus=           FALSE;
      m_bReadyToProcessARequest=    TRUE;

       //This is necessary if we wish to pop up a dialog box to user confirming
      the release of the Tuner
      //m_bTunerReleaseConfirmationFromUser= TRUE;

      m_hEvStopCheckingTunerAv= CreateEvent(NULL,FALSE,FALSE,FALSE);

      InitializeCriticalSection(&m_csPostMsgToProcessMessageWnd);

      m_WinMsg=RegisterWindowMessage( WINDOWS_MESSAGE_NAME );
      m_hEvWaitForProcessedMsg= CreateEvent(NULL,FALSE,FALSE,NULL);
      //
      // periodic attempts (60 secs) to acquire tuner when not owning it:
      // m_RequestState = STATE_REQ_WAITING_FOR_RESPONSE
      // or
      // m_RequestState = STATE_REQ_WAITING_FOR_TUNER_AVAILABLE
      //
      CreateTunerAvailabilityThread();

      // create the T.A. mutex in non-signaled state or
      // get a handle if mutex is already created.
      m_hTAMutex= CreateMutex( NULL, FALSE, MUTEX_NAME );
      if ( m_WinMsg==0 || m_hTAMutex==NULL || m_hEvStopCheckingTunerAv==NULL
      || m_hEvStopCheckingTunerAv==NULL )
      {
            // throw an error
            OutputDebugString("Error Condition\n");

      }
}

CTunerArbitration::~CTunerArbitration()
{


      // stop attempts to acquire tuner
      DestroyTunerAvailabilityThread();
      // wait for user feedback
      if ( m_hUserFeedbackThread!=NULL )
      {
            // thread is running, wait for user feedback.
            WaitForSingleObject(m_hUserFeedbackThread,INFINITE);
      }

      if ( TRUE==ReleaseTuner() )
      {     // send the good news around...
         BroadcastMessage( MKTUNERMSG(IC_TUNER_REQUEST,TUNER_AVAILABLE) );
      }

      DeleteCriticalSection(&m_csPostMsgToProcessMessageWnd);

      if ( m_hEvWaitForProcessedMsg )
      {
            CloseHandle(m_hEvWaitForProcessedMsg);
      }
      if ( m_hTAMutex )
      {
            CloseHandle(m_hTAMutex);
      }

}
```

```
BOOL CTunerArbitration::TunerOwned(void)
{
      BOOL  bRet;

      if ( !m_bOwnMutex )
      {
            // verify if mutex is owned by another app.
            if ( WAIT_TIMEOUT==WaitForSingleObject(m_hTAMutex,0) )
            {     // mutex is not signaled = tuner is owned by another app
                  bRet= TRUE;

            }
            else
            {     // mutex was signaled => tuner is not owned and is available
                  ReleaseMutex(m_hTAMutex); // restore the signaled state of
                  the Mutex
                  bRet= FALSE;

            }
      }
      else
      {
            bRet= TRUE;

      }

      // TRUE if mutex exist, otherwise FALSE.
      return bRet;
}

BOOL  CTunerArbitration::AcquireTunerWhenAvailable(void)
{
      BOOL bTunerAcquired;


      // try to acquire tuner, if not available we will wait for it
      if ( AcquireTuner() )
      {     // this obj now owns the tuner. no need to wait so go to sleep in
      idle mode ...
            m_RequestState= STATE_REQ_IDLE;
            bTunerAcquired=   TRUE;

      }
      else
      {     // tuner is owned by another app.
            m_RequestState= STATE_REQ_WAITING_FOR_TUNER_AVAILABLE;
            bTunerAcquired=   FALSE;

      }

      return bTunerAcquired;
}
```

```
BOOL  CTunerArbitration::AcquireTunerNow(void)
{
      BOOL bTunerAcquired;


      // try to acquire tuner, if not available we will wait for it
      if ( AcquireTuner() )
      {     // this obj now owns the tuner. no need to wait so go to sleep in
            idle mode ...
            m_RequestState= STATE_REQ_IDLE;
            bTunerAcquired= TRUE;

      }
      else
      {     // tuner is owned by another app.
            //
            // try to get it from other app.

            BroadcastMessage( MKTUNERMSG(IC_TUNER_REQUEST,TUNER_REQUESTED) );

            OutputDebugString("Broadcasting Tuner request message\n");
            m_RequestState= STATE_REQ_WAITING_FOR_RESPONSE;
            bTunerAcquired= FALSE;
      }


      return bTunerAcquired;
}

//------------------------------------------------------------------------------
//    private methods
//------------------------------------------------------------------------------
BOOL
CTunerArbitration::AcquireTuner(void)
{
      BOOL  bRet;

      OutputDebugString("Acquiring Tuner\n");

      // acquire mutex only if we do not already own it !
      if ( !m_bOwnMutex )
      {
            if ( WAIT_OBJECT_0==WaitForSingleObject(m_hTAMutex,0) )
            {     // mutex was signaled.
                  // this object now owns the mutex - now it's time to get the
                  tuner.
                  m_bOwnMutex=TRUE;

                  bRet =TRUE;

            }
            else
            {     // mutex is not signaled => tuner is owned by someone else.

                  bRet= FALSE;
            }
      }
      else
      {
            bRet= TRUE; //tuner already owned !
      }


      return bRet;
}

//v1.1
BOOL  CTunerArbitration::ReleaseTuner(void)
{
      BOOL  bRet;


      if ( m_bOwnMutex )
      {
            BOOL bMutexReleased;

            // Release Mutex
            bMutexReleased= ReleaseMutex(m_hTAMutex);
            m_bOwnMutex= FALSE;
            // Release Tuner
            ReleaseIntercastVideoStack();

            bRet= TRUE;
      }
      else
      {
            bRet= FALSE;
      }


      return bRet;
}

int
CTunerArbitration::ProcessMessage(HWND hWnd,WPARAM wParam,LPARAM lParam)
{
      int iRet=NOTHING;



      // keep a copy  of the hWnd that received the message (our hWnd)
      m_hMessageWnd= hWnd;

      if ( m_bOwnMutex )
      {

            // this object owns the mutex - and the tuner.
            //
            // since we own the tuner, only care about the REQUEST messages we
            receive


            if ( lParam==MKTUNERMSG(IC_TUNER_REQUEST, TUNER_REQUESTED) )
            {

                  // the tuner is requested by another app.
                  // respond according to this object policy :
                  // 1- ask a confirmation from the user,
                  // 2- release right away the tuner ,
                  // 3- keep the tuner regardless of the requests

                  // Denies any request if one is already being processed.
                  if ( m_bReadyToProcessARequest )
                  {

                        // indicate this obj is now busy with this request,
                        // and cannot accept other requests until done with
                        this one.
                        m_bReadyToProcessARequest= FALSE;

                        if ( m_RequestPolicy==ASK_USER )
                        {
                              if ( m_hUserFeedbackThread==NULL )
                              {
                                    // start a thread that takes care of
                                    obtaining user response.
                                    // -> avoid being trapped waiting for user
                                    response.
                                    DWORD dwThreadId;
                                    // pass params to thread
                                    m_wParam= wParam;

                                    m_hUserFeedbackThread= CreateThread
                                    (NULL, 0,

(LPTHREAD_START_ROUTINE)GetUserFeedback_static,
                                                      this, 0, &dwThreadId );
                              }
                        }
                  }
                  else
                  {


                        SendMessage( (HWND)wParam,
MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_DENIED) );
```

```
                        }
                }
                else if ( lParam==PRIVATE_IC_TUNER_RELEASE )
                {
                        m_bPostedCmdStatus= ReleaseTuner();
                        // cmd has been processed
                        SetEvent(m_hEvWaitForProcessedMsg);

                        //if policy is ASK_USER, we have to return LOST_TUNER
                        message
                        //      iRet=LOST_TUNER;


                }
                else if ( lParam==PRIVATE_IC_TUNER_ACQUIRE )
                {
                        // cmd has been processed
                        SetEvent(m_hEvWaitForProcessedMsg);
                }


        }
        else
        {


                // this object does not own the mutex - nor the tuner.
                //
                // since we do not own the tuner, we care about RESPONSE to our
                requests and
                // the TUNER_AVAILABLE request
                //


                if ( lParam==MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_ACCEPTED)
)
                {


                        if ( m_RequestState==STATE_REQ_WAITING_FOR_RESPONSE )
                        {
                                if ( AcquireTuner() )
                                {
                        // this obj now owns the tuner. no need to wait so go to
                        sleep in idle mode ...
                                        m_RequestState= STATE_REQ_IDLE;
                                        iRet= ACQUIRED_TUNER;


                                }
                                else
                                {       // unknown problem acquiring the tuner.
                        m_RequestState= STATE_REQ_WAITING_FOR_TUNER_AVAILABLE;

                        }
                  }
                  else
                  {
                  }
            }


else  if ( lParam==MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_DENIED))
            {
                  OutputDebugString("Tuner Request Denied\n");

                  if ( m_RequestState==STATE_REQ_WAITING_FOR_RESPONSE )
                  {     // the application does not want to release the tuner.
                        // wait for tuner to be available...
                        m_RequestState= STATE_REQ_WAITING_FOR_TUNER_AVAILABLE;

                  }
                  else
                  {
                  }
            }

            else if (
lParam==MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_WAIT_FOR_USER))
            {

                  if ( m_RequestState==STATE_REQ_WAITING_FOR_RESPONSE )
                  {     // a final response should follow soon ...
                  }
                  else
                  {
                  }
            }

            else if ( lParam==MKTUNERMSG(IC_TUNER_REQUEST,TUNER_AVAILABLE) )
            {
      //
      // Right now, this object will broadcast a TUNER_AVAILABLE msg only when
      it is destroyed,
      // so no need to protect against attempts to re-acquire the tuner after
      making it available.
      //
                  if ( m_RequestState==STATE_REQ_WAITING_FOR_TUNER_AVAILABLE
|| m_RequestState==STATE_REQ_WAITING_FOR_RESPONSE )
                  {
                        // tuner is available !! try to get it !!!!
                        if ( AcquireTuner() )
                        {     // yes !!
                              m_RequestState= STATE_REQ_IDLE;
                              iRet= ACQUIRED_TUNER;

                        }
                        else
                        {     // not able to get tuner ! continue to wait for
                              it ...
                        }
                  }
                  else
                  {
                  }
            }

            else if ( lParam==MKTUNERMSG(IC_TUNER_REQUEST, TUNER_REQUESTED) )
            {
            }

            else if ( lParam==PRIVATE_IC_TUNER_ACQUIRE )
            {
                  // tuner is available !! try to get it !!!!
                  if ( m_bPostedCmdStatus=AcquireTuner() )
                  {     // yes !!
                        m_RequestState= STATE_REQ_IDLE;
                        iRet= ACQUIRED_TUNER;

                  }
                  else
                  {     // not able to get tuner ! continue to wait for it ...
                  }
                  // cmd has been processed
                  SetEvent(m_hEvWaitForProcessedMsg);
            }

            else if ( lParam==PRIVATE_IC_TUNER_RELEASE )
            {
                  // cmd has been processed
                  SetEvent(m_hEvWaitForProcessedMsg);

            }

            else
            {
            }
      }


      return iRet;
}

BOOL CTunerArbitration::AcquireIntercastVideoStack()
{
      DWORD                   dwError=1;//assume error
      //attempt to acquire tuner

      m_OwnVideoStack= (dwError == 0);

      return dwError==0 ;
}


BOOL CTunerArbitration::ReleaseIntercastVideoStack()
{
      if ( m_OwnVideoStack )
      {
            m_OwnVideoStack= FALSE;
      }


      return TRUE;
}



BOOL CTunerArbitration::BroadcastMessage( LPARAM lParam )
{
      return SendMessage(HWND_BROADCAST,lParam);
}

BOOL
CTunerArbitration::SendMessage( HWND hDestWnd , LPARAM lParam )
{

      // if WHND = 0 , broadcast msg.
      hDestWnd= hDestWnd==0 ? HWND_BROADCAST : hDestWnd;

      BOOL bRet= (BOOL)PostMessage( hDestWnd, m_WinMsg, (WPARAM)m_hMessageWnd,
lParam );

      return bRet;
}

static void GetUserFeedback_static(PVOID pParam)
{
      CTunerArbitration* pObj= (CTunerArbitration* )pParam;
      pObj->GetUserFeedbackThread();
}

/*
No inter-thread protection required : only reading data from 'content' members
*/
void CTunerArbitration::GetUserFeedbackThread(void)
{
      int iYesNo;
      LPARAM      lParam;
      BOOL  bReleaseTuner;//assume we will release the Tuner

      if ( m_bTunerReleaseConfirmationFromUser )
      {     // requesting app has to wait for user response
            SendMessage( (HWND)m_wParam,
MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_WAIT_FOR_USER) );

            // ask user
            iYesNo= ::MessageBox(NULL,"Another application requests the usage
of the Tuner. Accepting this request will prevent WaveTop Receiver from
receiving data.\nDo you want to proceed with this request ?","WaveTop Receiver
received a request for the
tuner",MB_YESNO|MB_ICONQUESTION|MB_TOPMOST|MB_SETFOREGROUND);


      }

      // send command through main thread message loop

      bReleaseTuner=iYesNo==IDYES ;
      lParam=           iYesNo==IDYES ?
            MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_ACCEPTED) :
            MKTUNERMSG(IC_TUNER_RESPONSE,TUNER_REQUEST_DENIED)     ;
      if ( bReleaseTuner )
      {     // release tuner and mutex before sending ACCEPT response

            if ( PostMsgToProcessMessageWnd(PRIVATE_IC_TUNER_RELEASE,0) )
            {
            }
            else
            {
            }


            // now sit and wait for tuner to be available again ...
            m_RequestState= STATE_REQ_WAITING_FOR_TUNER_AVAILABLE;
            // to prevent the TunerAvailabilityThread to attempt re-acquiring
            the tuner
            // that has just been released!
            m_bSkipTheNextAttemptToAcquireTuner= TRUE;

      }
      // notify all users the tuner is now available.
      BroadcastMessage( lParam );
      // Ready to process another request
      m_bReadyToProcessARequest= TRUE;

      // this thread is now history ...
      CloseHandle(m_hUserFeedbackThread);
      // allow thread to be created again for another request.
      m_hUserFeedbackThread= NULL;
}
```

```
static void CheckTunerAvailabilityThread_static(PVOID pParam);

BOOL CTunerArbitration::CreateTunerAvailabilityThread(void)
{
      DWORD dwThreadId;
      // start thread, if not already running.
      if ( NULL==m_hTunerAvThread )
      {
            // start a thread that periodically attempts to acquire the tuner
            m_hTunerAvThread= CreateThread(NULL, 0,

(LPTHREAD_START_ROUTINE)CheckTunerAvailabilityThread_static,
                                    this, 0, &dwThreadId );
      }

      return m_hTunerAvThread!=NULL;
}

static void CheckTunerAvailabilityThread_static(PVOID pParam)
{
      CTunerArbitration* pObj= (CTunerArbitration* )pParam;
      pObj->TunerAvailabilityThread();
}

void CTunerArbitration::TunerAvailabilityThread(void)
{

      // perform check every 5 seconds
      while( WAIT_TIMEOUT==WaitForSingleObject(m_hEvStopCheckingTunerAv,5000)
)
      {


            if ( m_RequestState == STATE_REQ_WAITING_FOR_RESPONSE ||
                   m_RequestState == STATE_REQ_WAITING_FOR_TUNER_AVAILABLE )
            {
                  if ( !m_bSkipTheNextAttemptToAcquireTuner )
                  {
            // attempt to acquire the Tuner from apps that DO NOT comply with
            TA mechanism
                        // DO NOT perform official TUNER_REQUEST.

                        // AcquireTuner()
                  if ( PostMsgToProcessMessageWnd(PRIVATE_IC_TUNER_ACQUIRE,0)
)
                        {     // yes !! we now own the tuner !
                              m_RequestState= STATE_REQ_IDLE;

                        }
                        else
                        {
                        }
                  }
                  else
                  {
                        m_bSkipTheNextAttemptToAcquireTuner= FALSE;
                  }
            }
      }


      return;
}

void CTunerArbitration::DestroyTunerAvailabilityThread(void)
{
      // end thread if running.
      if ( m_hTunerAvThread!=NULL )
      {
            SetEvent(m_hEvStopCheckingTunerAv);
            CloseHandle(m_hTunerAvThread);
            m_hTunerAvThread= NULL;
      }
}



BOOL CTunerArbitration::PostMsgToProcessMessageWnd(LPARAM lParam,WPARAM
wParam)
{
      EnterCriticalSection(&m_csPostMsgToProcessMessageWnd);

      BOOL bRet= ::PostMessage(m_hMessageWnd, m_WinMsg, wParam, lParam);

      if ( bRet )
      {     // for until the window procedure has processed the message
            WaitForSingleObject(m_hEvWaitForProcessedMsg,INFINITE);

            bRet= m_bPostedCmdStatus;
      }
      else
      {
      }

      LeaveCriticalSection(&m_csPostMsgToProcessMessageWnd);

      return bRet;
}
```

CONEXANT™
What's next in communications technologies.™

**Further Information**
literature@conexant.com
1-800-854-8099 (North America)
33-14-906-3980 (International)

**Web Site**
www.conexant.com

**World Headquarters**
Conexant Systems, Inc.
4311 Jamboree Road
P. O. Box C
Newport Beach, CA
92658-8902
Phone: (949) 483-4600
Fax: (949) 483-6375

**U.S. Florida/South America**
Phone: (727) 799-8406
Fax: (727) 799-8306

**U.S. Los Angeles**
Phone: (805) 376-0559
Fax: (805) 376-8180

**U.S. Mid-Atlantic**
Phone: (215) 244-6784
Fax: (215) 244-9292

**U.S. North Central**
Phone: (630) 773-3454
Fax: (630) 773-3907

**U.S. Northeast**
Phone: (978) 692-7660
Fax: (978) 692-8185

**U.S. Northwest/Pacific West**
Phone: (408) 249-9696
Fax: (408) 249-7113

**U.S. South Central**
Phone: (972) 733-0723
Fax: (972) 407-0639

**U.S. Southeast**
Phone: (919) 858-9110
Fax: (919) 858-8669

**U.S. Southwest**
Phone: (949) 483-9119
Fax: (949) 483-9090

**APAC Headquarters**
Conexant Systems Singapore, Pte. Ltd.
1 Kim Seng Promenade
Great World City
#09-01 East Tower
SINGAPORE 237994
Phone: (65) 737 7355
Fax: (65) 737 9077

**Australia**
Phone: (61 2) 9869 4088
Fax: (61 2) 9869 4077

**China**
Phone: (86 2) 6361 2515
Fax: (86 2) 6361 2516

**Hong Kong**
Phone: (852) 2827 0181
Fax: (852) 2827 6488

**India**
Phone: (91 11) 692 4780
Fax: (91 11) 692 4712

**Korea**
Phone: (82 2) 565 2880
Fax: (82 2) 565 1440

Phone: (82 53) 745 2880
Fax: (82 53) 745 1440

**Europe Headquarters**
Conexant Systems France
Les Taissounieres B1
1681 Route des Dolines
BP 283
06905 Sophia Antipolis Cedex
FRANCE
Phone: (33 1) 41 44 36 50
Fax: (33 4) 93 00 33 03

**Europe Central**
Phone: (49 89) 829 1320
Fax: (49 89) 834 2734

**Europe Mediterranean**
Phone: (39 02) 9317 9911
Fax: (39 02) 9317 9913

**Europe North**
Phone: (44 1344) 486 444
Fax: (44 1344) 486 555

**Europe South**
Phone: (33 1) 41 44 36 50
Fax: (33 1) 41 44 36 90

**Middle East Headquarters**
Conexant Systems
Commercial (Israel) Ltd.
P. O. Box 12660
Herzlia 46733, ISRAEL
Phone: (972 9) 952 4064
Fax: (972 9) 951 3924

**Japan Headquarters**
Conexant Systems Japan Co., Ltd.
Shimomoto Building
1-46-3 Hatsudai,
Shibuya-ku, Tokyo
151-0061 JAPAN
Phone: (81 3) 5371-1567
Fax: (81 3) 5371-1501

**Taiwan Headquarters**
Conexant Systems, Taiwan Co., Ltd.
Room 2808
International Trade Building
333 Keelung Road, Section 1
Taipei 110, TAIWAN, ROC
Phone: (886 2) 2720 0282
Fax: (886 2) 2757 6760