R&D White Paper

WHP 136

May 2006

A Delphi component set for aaf

R. Storey

Research & Development
BRITISH BROADCASTING CORPORATION

BBC Research & Development
White Paper WHP 136

A Delphi component set for aaf

R Storey

Abstract

This note describes a set of Delphi Rapid Application Development components which wrap the aaf sdk[1] and provide a range of functions required for building media Applications.

The set includes components that allow aaf files to be created, edited, loaded and saved, including effects and a range of metadata types, without needing an in-depth knowledge of aaf. There are components for quickly setting up the data required for a legal aaf file and for running a multi-level undo-redo facility, as required by media professionals.

The set also includes components for replay of media files, playing a sequence of media files as a composition with transition and composition effects, displaying a complex multi track timeline, detecting shot-changes, pulling a set of key-frames out of a file, controlling the players and calculating and displaying timecodes.

The component set can replay mpeg2 files with frame accuracy and with high speed scrub-play, and can maintain audio video synchronisation when playing separate audio and video files. The aaf parts of the set conform to the Edit Protocol and stand a good chance of being readable by other Edit Protocol applications, assuming they use version 1.1 of the AAF SDK. The files will not be readable by applications using version 1.0.

The component set can be found at http://www.SourceForge.net/projects/aaf-edit-pack and is also available via http://www.bbc.co.uk/opensource/

Additional key words: AAF, Edit Protocol, mpeg, players, sequence players, timecode, shot-change, key-frame, scrub-play.

[1] www.sourceforge.net/projects/aaf

White Papers are distributed freely on request.

Authorisation of the Chief Scientist is required for publication.

BBC Research & Development
White Paper WHP 136

A Delphi component set for aaf

R Storey

BBC Research & Development
White Paper WHP 136

A Delphi component set for aaf

R Storey

1 Delphi Rapid Application Development

Delphi at its simplest, provide a means for linking together other people's work, in the shape of Components, into fully functional and efficient applications, using a relatively small amount of your own code.

It's major benefit compared to other RAD development environments is that it provides full source code for its component library and the library is a good example of a clean class structure. This makes it relatively easy to derive your own components if you need to provide specialised functionality. There are also a large number of Open Source components so, if you want others to be able to use your work, it is a good starting point for learning the more complex subject of component development.

One area not well covered by any development environment is media applications. These are normally closed source and not very helpful for researching new capabilities. There are some Open Source media players and editors but these run mainly on Linux. They use rather complex player libraries and are written mainly in C or C++. This component set is my attempt to open up the field of media application development for Windows and for those who are not fans of C++. It should be noted, however, that Borland C++ Builder and the newly launched Borland Studio 2006, can use components developed in Delphi.

2 List of Components

-  RSTimecodeEdit – A specialised TEdit that provides a readout of timecodes. It accepts times in frames, milliseconds, microseconds and DirectShow MediaTime. It can also be used to enter timecodes either as a full timecode with fill from the right, modification of one of the four timecode digit pairs or adding and subtracting a timecode from an existing value. It can also have its timecode modified by client code, for instance to set it to the nearest mpeg block start.
-  RSPlayerLib – This provides a media player that can play any file type for which your machine has direct show filters installed. It uses the Moonlight version 3 or Elecard version 4 mpeg decoding filter sets for decoding mpeg2 files and can provide absolute frame accuracy and rapid scrub-play, even when the file is longGoP and variable bit-rate. It uses file indexing to do this. If it does not find an index, it creates one an puts it in the media folder. It also provides a bitmap to display when video media is not available.
-  RSMediaSequence – This provides continuous replay of a composition of media files. It uses a pair of DirectShow filters for audio and video mixing. These filters, and therefore the component, provide a selection of 20 video effects including cross-fade and fade out and in, as well as level adjustment for audio files. The component provides a 'No Media' bitmap when a clip with no currently available video media is added to the timeline, and silent audio when a clip with no audio is added. The component currently plays only mpeg sequences either multiplexed with audio or with a separate .wav file.
-  RSPlayerControls – Provides a set of control buttons for the RSPlayerLib or RSMediaSequence Components. All that is required to control the players is to set the PlayerLib property of the control to the appropriate player. The button set can have its

member's visibility changed and its spacings set. It has mark in, mark out and play selections buttons and a capture image button as well as a full set of player controls. It a can also be set to auto size as a player is re-sized on the screen.

- RSPlayerTimeline – Provides a small timeline for displaying the position of the replay point for the players. It can display an In-Point and an Out-Point and its cursor can be dragged to change the player's position. All that is required to link to a player is to set its RSPlayerLib property to the required player.
- RSTimeline – Provides display and manipulation of a complex media timeline including effects, cut, copy and paste and clip dragging. It has a calibrated ruler for indicating position in the media sequence and the timeline can be stepped, stretched and moved in time. It supports dual track and single track video editing and has up to two pairs of stereo tracks and up to two pairs of mono tracks. There are also audio tracks that are locked to the video tracks for synchronised audio and these can be mono, stereo of five channel. It has an optional data track that can display various icons marking timeline events as well as showing text entries, and an optional draggable Scroll bar. The component links to a MediaSequence Component so that a clip or effect dropped onto the timeline or added in code can be displayed without user code needing to access the MediaSequence. There are also some Events that can be linked into an AAFEditAccess component to store the modified timeline ingredients into an aaf file.
- RSKeyFramePuller – Provides a series of bitmaps from a media file representing key frames supplied to it as a list. This list can come from an aaf file so that a series of clips from a bin or a project can be displayed in a list view component.
- RSThreadCutDetector – Detects the shot changes in a media file. The detector runs faster than real-time and can be used to split a file up into shot segments.
- RSThreadTimecode – Provides conversion between time values and timecode strings.
- RSIniFileManager – Allows quick setting up of an ini file to store application settings. Many of the components can use this one to store their settings. One Inifile can be assigned to as many components as you like provided you have a plan for what you will call the Keys and their entries. An example is the AAFUndoRedo Component that uses this component to store the details of a project's undo and redo stages when the application is shut down. It stores the values under keys derived from a project specific string so that you can retrieve the status of a project that you have not accessed for some time.
- RSLogFileManager – Provides a means to create logging data for an application. The log file can be set to use either the name and folder of the application or a separate name and folder.
- RSRegistryManager – Provides a simple means of saving and loading values into the Registry if that happens to be your preference. Personally I think it's a really bad idea since it means that moving an application from one machine to another without loosing settings is a headache.
- RSSerialPortAccess – Provides access to the serial ports on a machine. The Component provides a monitoring thread to allow the main application thread to avoid waiting for a serial port reply.
- RSLogIn – Provides a login window for the user to insert a user name and password. These can be checked for validity by user code.

-  RSGrid – A wrapper for the standard TStringGrid that provides column captions, a first column that displays Icons and a configurable number of timecode columns. This is used several times over in test applications for displaying comments, timecode breaks and keyframes.

The following aaf related Components are based on version 1.1 of the AAF SDK. Version 1.0 of the SDK will not be able to read them. This is not a bad thing since the Edit Protocol requires Version 1.1 and is the only specification currently available that allows precise placement of production metadata without vendors having to meet one another. Vendors should no longer use version 1.0 and in a sensible world, all would update their products to version 1.1 with no hesitation whatsoever.

-  AAFCreator – Provides a means to create a new aaf file. This Component includes functions for adding the basic ingredients that I have discovered while creating Bin, Logging and Editing files. There may well be many items missing. The current understanding is that the file will describe only one type of audio and video essence although there may be many instances. This is not a basic requirement for media applications and many are capable of mixing different types of media - but it simplifies the development. Many professional applications insist on only one media type. The Component also provides methods for specifying the media's video and audio properties and its Tape and Film properties where they exist. The Component can be used to create a basic Bin file that holds information about a piece of media. This file can then be built upon to add more metadata.

-  AAFLoggingAccess – Descends from AAFCreator and adds functions for creating, editing and deleting shot and timed comments, Key Frames and timecode discontinuities. The understanding behind this component is that the media comes from a single file source and the Logging application is used to create a valid programme file from it. The Component also provides functions for inserting, editing and deleting shot changes and for accessing the properties of the file's Video, Audio, Tape and Film properties where they exist. The Component can be used to build additional logging metadata into a file created by the AAFCreator component.

-  AAFEditAccess – Descends from AAFLogging Access and adds the ability to store references to more than one piece of source media, as well as using these as sources for clips in the timeline and a range of video and audio effects. Some of the comment and key frame capabilities of the LoggingAccess are reused in this component to provide metadata referring to a newly created composition. The component goes to some lengths to ensure that, when additional clips are added that are derived from a given source media object, then the file will contain one and only one source media object. This sounds simple but for some reason, it wasn't.

-  AAFProductIdent – AAF insists that a record is kept of all of the applications that have accessed the file. This component provides a simple means for providing this information for a given application. Each Access Component has a ProductIdent Property which must be set to the value of an AAFProductIdent. The Component provides a means for using the Executable's name and version or a value set in the Object Inspector. Sadly, when you select UseExeName, the Object inspector will show the properties for the IDE since the executable is not yet launched but all will be well when the application is running.

-  AAFRawStorageMemory,  AAFRawStorageDisk,  AAFRawStorageCacheDisk; - Provide access to Storage either in memory, on disk or on a cached disk. These would normally be used for storing media within the file. They appear to work but I have not tested them since all of my work has assumed external essence located using an EssenceLocator.

-  AAFUndoRedo – This is a rather neat component that it provides a means to store multiple versions of a project as it progresses. The storage is achieved by storing an aaf file

with an added index in the filename. The aaf files are relatively small if you do not store essence in them. The component provides a tree of project stages and allows backtracking through a project. A back track will produce a branch in the tree that will normally be collapsed so the user can keep several different versions and access them from the tree view. This provides a very useful facility where users are trying out different combinations of media clips to identify which is the best. It avoids the need to keep several different versions of a project.

-  AAFLogger – This component collects all of the non GUI related functions required for a logging application into one class. This approach allows quicker development of a logger with a different design of GUI and keeps the Form code free from non Form related functions. It assumes that you already have a TAAFLoggingAccess and a TPlayerLib in your application. You may also have a TRSTimeline if required. If you do not have a TRSTimeline then any function that requires one will fail. It supports loading, saving and closing a logging file that points to external media; handling of shot and timed comments and Key Frames; and insertion, removal, time adjustment and naming of shot changes with timings handled correctly for logging. It also supports accessing the file and derivation Mob properties included in a file which determine to whom the media belongs.
-  AAFEditor – This component collects all of the non GUI related functions required for an Editor into a single class. This approach allows quicker development of an Editor with a different GUI design and keeps the Form code free of non Form related functions. It assumes you have a TAAFEditAccess, a TRSMediaSequence and a TRSTimeline. You may also have a TAAFLoggingAccess if you want to load the contents of an existing aaf file and have the derivation Mobs transferred into the editing project file. It supports loading, saving and closing an editing project file that points to external media; handling of shot and timed comments and Key Frames; naming of shots; handling of a set of video and audio effects; setting of video flip and flop properties; and loading of clips for which audio and/or video is not yet available. It also supports accessing the File and Derivation Mob properties used in a project, including at what times and for how long they are used. This is gold dust for professional broadcasters.

There are also some additional, Mozilla licensed, third party components added for application development:

-  PJDropFiles – Provides a simple method for retrieving lists of files dropped onto a form. This is used to drag files from another windows application onto you application. Drag and Drop is a lot easier for the user than messing around with Open Dialogs.
-  MRUList – Provides a drop down list containing a selection of recently accessed files.
-  PersistentPosition – Restores the previous form positions when an application is re-started.
-  RunOnce – Prevents multiple instances of a application. This one has some additional properties to allow the first application to know a new one has been launched so that it can, for instance, steel its parameter list.

3 Edit component descriptions

These components provide media specific functionality for building media based applications. Their functions, procedures and properties are described in a Delphi Help file included in the component pack. The following text outlines their functions and some information required to use them.

3.1 RSTimecodeEdit. (RSTimecodeEdit.pas) 

This visual component allows display and user entry of timecode information. It is based on a TEdit component and supports most of TEdits's properties and methods. It provides direct input of timing information in different formats. These are Frames, milli seconds, micro seconds and Microsoft Media Time, which is in units of 100 nano seconds. The control also has an EditRate property which allows it to translate time and frame information into timecode. This property can be either a public TRational which has a numerator and denominator value or in the property inspector the numerator and denominator values are provided. The value defaults to 25:1 for Pal video.

The control also supports user input of timecodes. A single click on a timecode pair will select it for data entry. The colour of the entered text changes according to whether the value is a valid or not, being the UnconfirmedFontColor if the value is valid but not yet confirmed or the ErrorFontColor if the value is not valid, 75 seconds for instance. The new value is confirmed by pressing Enter and digits can be removed one at a time by pressing backspace.

A double click on the control selects the entire timecode for data entry and the same rules apply. It is not necessary to enter leading zeros before being able to confirm the new timecode.

If the plus or minus keys are pressed while the control is focussed, it goes into incremental entry mode signalled by a +:00 or -:00. When in this mode the first two digits entered are in frames and can be up to a value of 99. If more digits are entered, the input is assumed to be a timecode and has to be valid. The new value is confirmed by pressing Enter and digits can be removed one at a time by pressing backspace.

The control also allows the user to step the values of any of the four timecode couplets using the arrow keys. This mode is triggered by pressing the left or right keys to select a couplet. The value can then be stepped by clicking the up or down keys. The control calls its OnStep event which allows client code to modify the value inserted into the timecode edit control. This is useful if you are for instance, attempting to define the starting frame of a section to be cut out of an mpeg long GoP sequence which, if you are not decoding and recoding, will normally begin at a position related to the nearest I-Frame. The OnStep event passes the new timecode as a var property along with the Step value which is the number of frames by which the the control has been stepped. Client code can then return the required frames value.

3.2 RSPlayerLib (RSPlayerLib.pas) 

This visual component is a media player that specialises in playing long GoP mpeg with frame accuracy and good audio-visual synchronisation. The synchronisation works well when playing either multiplexed or separate files. It appears on a form as a TPanel upon which the video part of the media is played. The audio is played through the PC's sound system.

The control builds a DirectShow graph which it populates either with specified filters when some feature not provided by most DirectShow filters is required, such as frame accurate positioning and rapid file seeking for longGoP mpeg2, and with whatever filters are installed on the machine by using the auto-connect features of DirectShow. For mpeg I have included the possibility to use four sets of mpeg decoder filters. The first is Elecard version 2 filters, which do not provide accurate positioning. The second is Moonlight-Elecard version 3 filters, which are now difficult to find since Moonlight have gone into liquidation, but provide frame accurate positioning and good file seeking. The third is Elecard-MainConcept filters which also provide accurate positioning. The fourth set of filters are an open source set which do not yet provide frame accurate positioning and do not yet have an Open Source de-mulitplexer. It follows that the positioning for mpeg with the required filters is guaranteed whereas for other file formats and filter sets, it will be whatever the installed filters provide.

The control supports full screen replay, setting of in and out points, replay of a portion of a clip as though it were an entire clip, superposition of text on the display area when using older renderers,

capture of frames to bitmaps or images, indexing of mpeg files that do not already have an index, selection of next or previous edit points (rather simple in a player), and an assortment of player controls including stepping forward and backwards by single or multiple frames and rapid replay.

As mentioned above, the mpeg replay uses specialised sets of decoder filters. These are currently available from Elecard MainConcept but there is also support for Moonlight Elecard filters that may well be installed on many machines. These filters allow indexing of an mpeg file to find the file offsets for a given frame precisely instead of searching for them from a guess based on the file size, which can be stupendously wasteful of cpu power, and network resources if the file is remote.

3.2.1 File length notification

Mpeg was designed for efficient media delivery and so does not include a property that defines the media length in frames. I can only think this was in the interests of bit stream efficiency, although the few bytes that would be required to define the media length would surely not have mattered. The upshot of this is that most mpeg media players are forced to guess the number of frames in a file from its bit-rate and length in bytes. This can be a very inaccurate guess, especially if the file is variable bit-rate and long GoP, both of which are required to get best video quality for a given file size.

Sadly, to a media professional, the need to find a given frame is absolutely essential so we either have to count them by parsing the file, or parse the file in advance to produce a smaller list of indexes which can be interrogated quickly. The latter is the approach taken in this player since indexing a file can be done when it is captured and so will not cause a operational delay for the user. Parsing the file each time it is opened, however, would provide an operational delay whenever the file is opened, which would soon become irritating. The player can, however, create an index file when a media file is opened, if no index file already exists. The media will still play but will not be seekable quickly or accurately until the indexing is finished. The index file is stored in the media folder, provided it is writable, ready for the next time the media is opened. If the media folder is not writable, the index file will be stored in the player's index cache folder

No other file types are any better of course although the MXF file wrapper for mpeg does include index tables that allow the length and frame offsets to be determined quickly.

The player provides a property called MediaLength that provides an accurate value for the number of frames in a sequence once it has been indexed. If the indexing has not been completed, it will provide only an approximate value. The Player provides Events to tell client code when it thinks it needs to index a file and when the indexing has been completed. These are OnIndex and OnFileInfo respectively. The OnIndex Event repeats and delivers a progress value which is a SmallInt value. This can be used to drive a non duration measuring progress indicator, since the player does not yet know how long the file is. The OnFileInfo Event is fired when the indexing is completed and Delivers an instance of a TMediaFileInfo structure containing the accurate values.

3.2.2 Rapid scrub play.

Another feature prized by media professionals is the ability to scan from one end of a file to another quickly while being able to view its contents. This is essential for finding content in a large file and just plain nice to use. This means that the player refresh rate must be high, preferably in excess of 25-30 frames per second, and the position between images must not be limited. If the file is not indexed but is local, this is not too much of a problem on a fast PC since the repeated approximations to get the correct file position will happen rapidly, albeit at the expense of considerable cpu power. If the file is on a remote file system though, the update rate can fall to less than a frame every few seconds without indexing, which is unacceptable.

If the file is indexed, however, there will only be one correct file offset for each frame that is sought which makes local file scanning faster and less cpu intensive. Remote file scanning, however, becomes enormously quicker and the reduction in network loading has to be seen to be believed. Speeds in excess of 50 frames per second can be achieved over a loaded network, which means the player is faster than its user's perception.

MXF indexed files should also be almost as good but I have not yet found an efficient enough mxf splitter filter to be able to demonstrate this.

3.2.3 Media Positioning

An indexed mpeg file can have its position set precisely using this player. There are two basic types of position setting provided, precise and approximate. Precise sets the position to its exact value and approximate sets it to the nearest I-Frame. This differentiation is helpful when playing long GoP files since, if a player accesses say a B frame at the end of a Group of Pictures, the decoder has to decode the starting I-Frame plus all of the intervening P Frames before it can decode the B Frame that it actually wants to display. When a user is scanning through media at 500 frames per second they will not be interested in what frame type is shown. The player therefore uses precise positioning when it is required e.g. for stepping or setting a position from a timecode edit, and approximate where it is not required such as during rapid seeking, when stepping forward and backwards by a GoP length and when doing fast replay.

3.2.4 Positioning of the Media replay.

The component can alternatively position its replayed video onto another TPanel Control. This might be useful in an application where the same player component is used to perform, say media replay and setting some in an out points for separating out a section of video. These two operations are mutually exclusive, since only one form can be focused at once, and so can be done using only one DirectShow Graph instance. You will have to use the CaptureToBitmap Method to capture a bitmap to display over the disabled TPanel control.

3.2.5 Adding Text to the player image.

This was originally done using a DirectShow8 video renderer and its ColorControl features. This function has I believe been removed from the DirectShow9 renderers and you are now supposed to use a separate input pin on the renderer. I have not done this but you have a property called UseOldRenderer that forces the player to use an older renderer. Note you cannot simply stick a label on top of the player since DirectShow uses the Handle of the TPanel to write directly to the screen buffer.

3.2.6 Controlling the Player

You can use either the methods provided by the player driven from your own player buttons or you can use the TPlayerControls component. Using TPlayerControls allows you to set up the relationships automatically by simply setting the control's PlayerLib property to your TRSPlayerLib instance. TPlayerControls allows you to set which controls you want to have visible and includes buttons for setting and removing in and out points, playing a selected section of media and capturing the current frame to a bitmap or Image. TPlayerTimeline allows you to show a small media timeline with a cursor to indicate the player position. It too has a PlayerLib property that sets up all the relationships for you. It has a mark in and mark out icons that appear and disappear when you click the mark in and mark out buttons. You can also drag the player time position by dragging the timeline cursor and, once a mark in or mark out has been set, you can modify its position by dragging over the edge of the selection bar.

3.2.7 Loading media into the player

This is done using OpenPlayer, which allows you to specify a video filename, an audio filename if there is one and an index name. If you should have an audio filename and do not supply one, you will not get any audio. However, if you fail to supply an index filename and the media has an mpeg file extension, the player will attempt to index the file and, if it succeeds, it will attempt to store the index file in the same folder that the media occupies. This may not be what is required for a professional media asset management system but in any case, an MAM could look after generation of media index files itself. Alternatively, if the media folder is not writable. The index

file will remain in the player's index folder so then you only have a name conflict problem to sort out.

3.3 RSMediaSequence (RSPlayerLib.pas) 

This visual component is similar to the RSPlayerLib but allows you to play a collection of media files strung together in a timeline. The component currently handles separate m2v and wav files or multiplexed mpg files for combined audio visual replay or just video or audio. Management of audio-visual synchronisation is done by the component except when you want to modify the lengths of audio edits to make them different from the video – once you have done this, you are responsible for correcting any timing errors that are introduced by your actions. The effects are calculable but you will need a clear head to avoid making mistakes.

The component requires the RSVideoMixer.ax and RSAudioMixer.ax DirectShow filters to be installed and registered on your machine. It also requires a frame accurate set of mpeg decoding filters. Replay from the start of the media will probably work without frame accurate filters but audio/visual synchronism will be lost as soon as you position the player.

3.3.1 Setting of audio/video mode

The component can handle video and audio, just video or just audio media sets. This behaviour is set using the VideoMixer and AudioMixer boolean properties. Once a clip has been added to a sequence you cannot then change the mode of operation of the Media Sequence. The Media Sequence currently expects only a single audio file or multiplexed stream. This can be mono or stereo (it may work for multi-channel but this has not been tested).

3.3.2 Setting the graph properties

TimeFormat changes the time format of the component's interface for all of the times represented as Int64 values. The options are Frames, milliseconds, microseconds and Microsoft's MediaTime format. Any function that uses Frames remains in Frames.

The filter can use a VMR renderer or can be prevented from doing so if you want to overlay components of the image, which doesn't work for a VMR. You can also prevent the MediaSequence from using YUV2 format if you want. If you do so, the mixer will use an RGB format determined by your graphic card settings, which if it is set to 24 or 32 bits, is much less efficient. You may only notice this on complex effects.

If your decoder set supports half resolution replay, you can set this using the HalfResolution property. This property should not be set once you have a clip in the graph since there are no measures for updating all the graphs and the mixer will not replay files of different image sizes.

You can examine the Filter Graph by setting GraphToRot to true and picking it up in GraphEdit. This is useful for diagnosing failures to replay. The RSPlayerLib's property page will list all of the filters available on your machine but there is no property page for the MediaSequence.

3.3.3 Getting notification of the video cache status

The video mixer uses a cached output pin to prevent replay hesitation when you PC is busy doing something other than replay. It also helps to avoid hesitation when doing complex effects. You can link this to a visual component using the OnCacheUpdate event which provides you with notification when the number of frames cached changes. This is not very useful but might interest technical users.

3.3.4 Getting notification of completion of the timeline replay

The OnCompleteEvent fires when both the audio and video replays have completed. His notification can be used for resetting player controls to stopped once the player has reached its end. It is also a useful diagnostic since, if your replay times are badly broken it may not get fired.

3.3.5 Getting notification of the replay position

There are two events for this. OnTimecodeUpate notifies the current frame position and can be used to drive a TRSTimecodeEdit component. The audio time is provided by OnAudioTimeUpdate. This one will be confusing for users since audio replay runs a second or so in advance of the video and is synchronised by the audio renderer which appears to cache four samples ready for replay.

3.3.6 Adding Clips to and Removing them from the timeline

Clip sets can be added to the timeline using the PrePend, Insert and Append functions. These require a video and an audio filename, a start time and a stop time. If you have a multiplexed file then this is input as the video filename and the audio filename is left empty. The media sequence assumes that separate video and audio files are from the same capture and have the same start times with respect to the source media, and equal lengths. If you supply a later start time than the media allows you will be informed that the source media is not long enough and the sequence should recover safely. RemoveClip removes a clip from the timeline. RemoveClip also allows you to discard the effect at the start of the clip or move it to the start of the following clip.

3.3.7 Cutting, Pasting and copying Effects

This is done using the PrePend, Insert, Append and Delete functions but you need to set a property called EffectPosOnPaste to true or false. If you insert a clip between two clips that already have an effect, then if EffectPosOnPaste is true the effect remains tied to the start of the following clip. If it is false, the effect will be transferred to the start of the inserted clip.

3.3.8 Adding and removing video effects

There are separate functions for setting video and audio effects. For these functions to work you must have the AllowEffects property set to true. If you attempt to set it to false once there are effects in the timeline the setting will fail.

The setting of a video effect requires a TMixerEffect parameter and a length. The effect parameter is listed in the code and offers one of twenty built-in effects plus meNone which is used along with a zero length to remove an effect. The indexing assumes that the effect goes at the start of the indexed clip so an index can legally be out of range by one, referring to an apparently non-existent clip one beyond the end of the clip list (The mixer filter uses a dummy clip to hold this value). This applies the effect to the end of the last clip.

You can find the current effect at the start of a clip using the GetVideoEffect function and the current clip times using GetVideoClipTimes. If you change the effect type or length without using a true value of AdjustAudioEffect or after applying your own modifications, then you will first need to query the current effect and clip lengths to be able to work out what effect and clip lengths are required to retain audio/visual synchronisation.

When you add a video effect, you can also ask the function to include a matching audio effect, the type of which is set using the AudioEffect value and comes from a short TAudioMixerEffect enumeration. If you choose not to use this function then you will need to ensure that the audio effect length and clip times you use are calculated to ensure correct audio/visual synchronisation.

3.3.9 Adding and Removing audio effects

Audio effects are handled using the SetAudioEffect function which again requires a TAudioMixerEffect value and a length. Again you need to have the AllowEffects property set to

true. An audio effect can be removed by using and meaNone value and a zero duration. You can find out what the current audio effect is using GetAudioEffect. This together with clip times got through GetAudioClipTimes allows you to fully determine the layout of the audio tracks before applying your modifications.

3.3.10 Changing clip times

The audio and video clip start times with respect to their source media, and their durations can be set using SetVideoClipTiems and SetAudioClipTimes. There is no tying of audio times in SetVideoClipTimes so it is down to you to ensure the changes retain Audio/Visual Synchronisation.

3.3.11 Getting the timeline duration

This can be found using the Duration Property. It changes whenever the timeline is modified and is a read only property. To change the duration you must modify the timeline. There is also a OnVideoDuration update property that notifies what the new video duration is and an OnAudioDuration property that does the same for audio. This can be used to update an application to see, for instance, how far short of a required time your composition is. It can also be used to determine what, if any, are the differences in audio and video duration.

3.3.12 Linking clips to user interface components

The media sequence creates classes to manage the DirectShow filters for each file set used in the timeline. References to these classes are stored in the mixer filter's UserData property which is stored separately for each clip in the timeline. If your application needs to know which class is used for a particular clip then you can use the GetClipData property which returns a Pointer that should be cast as a TClipItem. This value will change as the timeline is modified since the MediaSequence needs to ensure that a given file does not need to be read twice during an effect . It does this by always ensuring that consecutive TClipItems using the same source files always alternate along the timeline. In short, if you need this data then don't store it but access the property when you need it.

3.3.13 Finding the timings of a given clip

These values can be read using the GetClipDetails which allows you to get the media start and stop times, the timeline start and the video mixer effect properties for a given clip index as well as a reference to the class providing the replay. I have not provided a means to get the audio timings.

3.3.14 Finding the clip index for a particular timeline time

This can be done using the GetClipIndicesAtTime function. This returns two indices for a given timeline time. The first is the index of the first clip at that time, the second will be the same if there is no effect or the following clip if there is an effect at that time.

3.3.15 Setting and getting audio clip levels

The audio filter offers the ability to set the audio replay levels throughout an audio clip. These can be set separately for the right and left channels if the media is stereo. I've not checked what happens if it is mono and the function will not work for multi-channel sound. The Levels are initialised to four values when the clip is inserted. The initial levels are $0000 at the start, $1000 one time interval into the clip and one time interval before the end and $0000 at the end. The time interval defaults to a frame period – apologies to audiophiles. This avoids popping at audio butt edits.

You can modify the levels by changing all but the first and last levels. You can also add your own levels to the clip at a given clip time. If you add a different level value then the audio gain will move to that new level progressively and in a linear fashion. When you add a new level, the function

returns the index of the new level. A level can be removed using RemoveLevel provided you have its index.

You can enumerate through the levels using an iterator that is initialised to a given clip using ResetEnumLevels, you can then iterate through the levels in that clip using NextLevel which returns the time and values of each level. This approach can be used to provide a display of the levels within a clip on a timeline component.

If you want to adjust a level from a user interface component then you will need to find the index from a time value which will generally not be precise. This can be done using GetLevel which allows you to find the nearest level within a given threshold. The idea here is that you click a clip on a timeline which provides you with an approximate position, you then decide on a threshold that makes sense compared to the length of the clip and the user's ability to position the mouse. You then call the GetLevel and if there is a level within the threshold, you get the ClipIndexes of the first and second clip if there is an effect there, plus the indices and the levels of the right and left channels, you are then in a position to respond to some user action that sets the levels

3.3.16 Control of the playing properties

The component provides the same player controls as the RSPlayerLib component except for the SubClipPlay function which does not make sense for an editing timeline. You can however select a start and stop time and play the selected portion of the timeline. The PreviousItem and NextItem calls are also a bit more interesting since the media sequence now has many more edits and effects that can be stepped between.

There are some optimisation controls. FastPlayInterval is the time between steps when the player is in fast play mode. It avoids the player asking for new positions before the previous one step has completed. Whether this helps or not depends on the decoding filters used. FastSeekThreshold determines the number of frames beyond which the player uses approximate positioning. SeekOptimiser is meant to limit the frequency of positioning updates when the user moves the player manually.

3.3.17 Getting and setting player positions

You can get the current player position using CurrentFramePos which returns the position in Frames or Position which returns the position in the chosen time format. There are a range of functions for setting the position. SetPosition allows you to set the player position to a value in the chosen time format. It examines the new position to determine if it differs from the last position and by how much. If the difference is large it uses approximate time settings which sets the first clip to the nearest I-Frame. If there is an effect there then the second clip is set precisely.

SetPositionAbs sets the position precisely to a value in the chosen time format. It is a good idea to call this one when the user stops moving the player position otherwise the audio and video times are very likely to differ, which will cause replay problems. SeekRelativeAbs sets the player to an absolute value between 0.0 and 1.0 and can be used by a visual component whose dimensions represent the media time. SeekToFrame sets the position precisely to a give frame value and can be used by a visual component that works in frames such as a TRSTimecodeEdit.

3.3.18 Setting the NoMedia bitmap

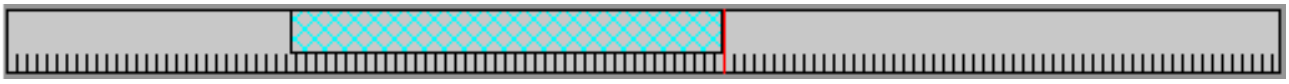
When the filter plays a clip for which there is currently no media files, it replays silence for the audio and displays a black image for the video. The black image can be modified using a bitmap which you can provide through the NoMediaBitmap property. The bitmap can be of any size but if it is smaller than the video image, it would need a black background to avoid its boundaries being visible. The bitmap is copied and so can be freed once the property has been set.

3.4 RSPlayerControls (RSPlayerLib.pas)

This visual component provide a set of media controls that can be used to build media and timeline players. You can link an instance to an RSPlayerLib or RSMediaSequence by setting its PlayerLib property. This makes all of the required links between the two components using protected functions so that you can still use all of the player and controls events for your code. The control allows you to set the visibility of the buttons, and their dimensions and spacings. You can also enable or disable buttons using the EnabledButtons property. The Buttons can be set to be normal SpeedButtons or Flat SpeedButtons.

The control can be set to AutoEnable its buttons when its linked player has media loaded. If this property is not true, then users may be a little confused when they click, for instance, the Play button and it does not remain down, since there is no media to play. The control also includes a 'capture' button which, when used with an RSPlayerLib causes the PlayerLib to either capture to a bitmap or file or call a custom event. This allows you to use this button, for instance, to set the position of a key frame.

3.5 RSPlayerTimeline (RSPlayerLib.pas)

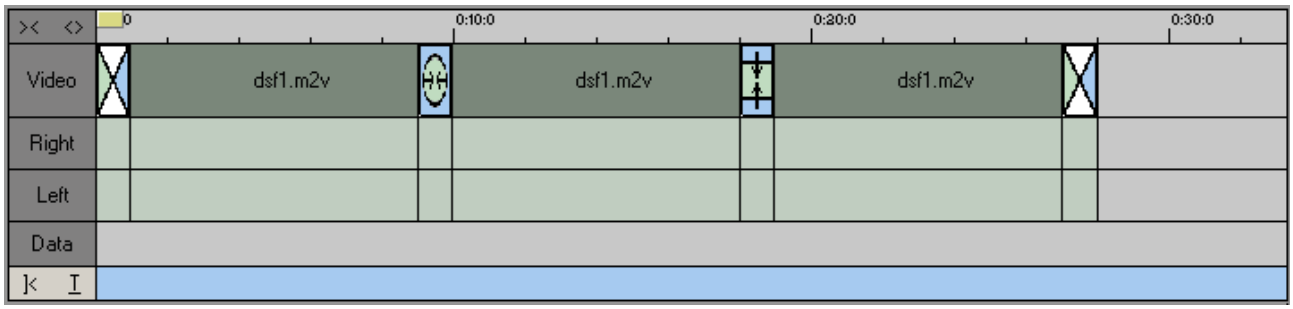


This visual component provides a small timeline that can be used for a media player or a sequence player. You can link an instance to an RSPlayerLib or an RSMediaSequence by setting its PlayerLib property.

You can move the player's position by dragging the timeline cursor, shown here in red. You can also set the position by clicking in the timeline. You can move the in and out points by clicking on the edge of the start or end of the selection box shown here checked in blue.

If you have a PlayerControl connected, then setting the in and out points using the buttons sets them for the player and also makes the selection box visible. As the player plays, the timeline cursor moves accordingly. If you use the mouse events on the player you can also make the player move by dragging on the player's image. I have used this to make the image dragging match the timeline dragging but you could also use it to change the replay speed.

3.6 RSTimeline (RSTimeline.pas)



This visual component is a big one and was the first one I built. It provides a means to display and manipulate the clips in a timeline. Clips and effects can be added to the timeline in code or by dragging and dropping TSourceClipItem or TClipItem drag objects. Dragging from the timeline to a player can also be used to modify the start and stop times before dragging back to the Timeline. A Guid is attached to each clip which allows the timeline to identify if a clip is dropped on top of itself, when its start times and duration replace the existing ones or if not, are simply inserted at the dropped position.

Clips can be selected singly or as consecutive groups if the AllowSelectByMouse property is true and dropped elsewhere if the AllowInternalDrag property is true, either as a copy or a cut operation. If AllowEffects is true and a clip is dragged using the right mouse button, an effect is inserted between the two clips and its length can be set by right mouse button dragging.

The Timeline has a MediaSequence property that if set, transfers the clip file properties and times to the media sequence. The intention was that all of the manipulations on the clips would be linked in this way but I have not completed all of the code for this yet.

The Timeline can support the display of bitmaps that describe the effect properties as shown in the graphic

The Timeline has a ruler whose scaling can be adjusted through code or in steps using the contract and expand buttons at the top left.

The Audio timeline can display clip levels stored in an internal list for each clip. These have to be managed in code and tied into the levels stored in the media sequence.

The video tracks can be either a single track as shown in the graphic or can be a dual video and effects track arrangement. The audio tracks can be mono, stereo or five channel for the base tracks that are tied to the video. There are also up to two sets of stereo tracks and two mono tracks. The type of audio track is configurable. The data track allows display of text and icons not related to clips. These can be used to display spoken text or the presence of timecode breaks etc..

There is also a scroll bar track that allows the user to move the view around inside a long timeline.

The timeline cursor can be linked to a player so that it drives the player position and can also driven by the player position if the player is moving under independent control.

All of the component track and clip colours, heights and visibilities can be configured in the Object Inspector. And there are separate PopupMenu items for the Data and Media Tracks. The timeline has its own popup menu that offers the ability to set the timeline origin back to zero and to set its time scale to one of a preset list of values. These functions are also duplicated by buttons at the lower left that set the origin to zero and fill the timeline if the Control key is down, or provide a popup list of time scalers.

The unit provides an assortment of other classes that may be useful when using the timeline.

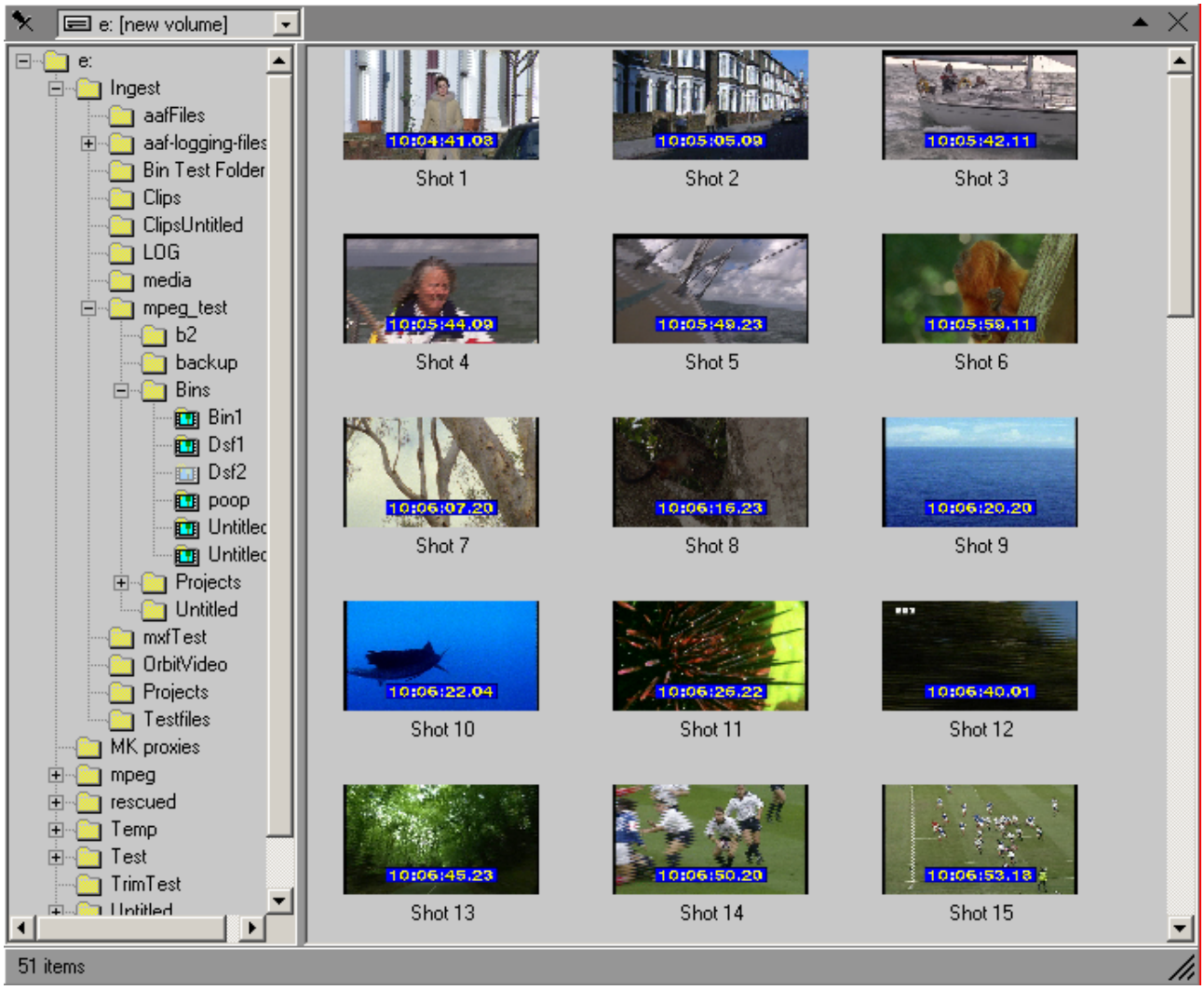
3.7 RSKeyFramePuller (RSKeyFramePuller.pas) 

This non visual component provides a means to extract bitmaps representing times supplied to the component in a TInt64List parameter. The idea behind this component is that you provide it with a media file and a list of key-frames pulled from an aaf file and it provides your visual component with a set of bitmaps that you can display in a visual control. An example of a file browser is shown below. This shows a tree view that has an aaf file opened and a list view that shows images of each of the clips within the aaf file.

The component runs through the media file on a separate thread and delivers the bitmaps to the application through a synchronised event. The component uses windows messages to link from the detection graph to the component so client code need not do any thread management.

The component provides the ability to set the Priority of its detection thread, set the bitmap dimensions and whether it should publish its graph to the Running Object table. The component has an IndexName property that provides the indexing file to allow it to get accurate positions for the images. It does not scan for the index file when you provide it with a media file so you need to provide this data your self.

The component can also pull a single image from a file by using the GetKeyFrame function which is single threaded and waits for the bitmap to be generated.



3.8 RSThreadCutDetector (RSThreadCutDetecotr.pas)

This non visual component houses a filter graph that does cut detection. It requires the DirectShow filter CutDetect.ax to be installed and registered on your machine. The component can support display of the video while it does its shot detection by assigning it a TPanel to use as a display.

It can be configured to use a media clock to force it to run at normal speed. If it does not use a clock then the detection runs at the maximum speed possible. This will involve more than one thread and may span more than one processor in a multi-processor machine, since DirectShow is multi-threaded.

Its default operation mode is to use only the luminance component of the video but you can make it use the Colour components too for a more ‘complete’ but slower result. I say complete because it may be that there are some shots that are very similar but differ in colour only, rare but possible. The detector will default to using YUV2 in its media decoding but can be forced to use an RGB format by setting AllowYUV to false. When you do this, the format used will be determined by the graphics card setting if you have a display connected, otherwise it will be RBG32.

The component provides a diagnostics output if the Diagnostics property is true. The diagnostics are delivered by an OnDiagnostics event that delivers four parameters. These are MAD which is a frame difference signal, Threshold which is the level above which the shot detection signal must pass to qualify as a shot change, CutSignal which is the value of the shot change detection signal

and StdConvCutSignal which is a value designed to detect one frame cross-fades as often happens in standards converted material. When Threshold is exceeded you get an OnShotChange event which delivers the Frame Number and the probability and length of the shot change. You can also keep track of the detector's progress if you already know the length of the media by using the OnCurrentFrame event, which delivers the value of the frame just processed. There is also an OnComplete event that can be used to stop and clear the detector when it has finished.

The detector can be Started Paused and Stopped without loss of timing accuracy. The filter also supports the IMediaPosition and IMediaSeeking interfaces so the replay can be positioned by a filter graph. Once it has been positioned, the reported frame position registers the frames from the point to which it was positioned, so some care is needed in using the results.

The component also includes setting of the mpeg decoder filter set as in the media players. If the filter set is frame accurate, you can ask the Graph to tell you what the time is when you stop moving the replay position, which should allow you to get an accurate frame number.

3.9 RSThreadTimecode (RSTimecode.pas) 

This non visual component provides conversions between different time formats and can be called safely by different threads. It handles times in Seconds as a double; frames as an integer or an Int64; MilliSeconds as an integer or Int64; and times as Microsoft MediaTimes as an Int64. It also provides timecodes as a TTimecode record. It provides conversions between these time formats and also provides timecode strings of the form HH:MM:SS:FF.

The unit also provides a non thread safe class called TRSTimecodeCalc that provides a variety of different conversions.

3.10 RSIniFileManager (RSIniFileManager.pas) 

This non visual component provides easier access to an ini file than using the Standard TIniFile class. It provides a means to name the file after the application e.g SimpleLogger.ini. If this option is chosen the file will also use the application folder for the location of the IniFile, otherwise it uses the IniFilePath setting in the Property Inspector. The IniFile has a LazyWrite option which if true, holds the file contents in memory until the application closes. If it is false, the contents are written whenever an new line is written to the component.

The component provides properties for accessing boolean, integer, Float, DateTime, Date, Time and string values. Every entry has a Section setting and a Name. You can also read a list of Sections as a TStringList, Read the contents of a Section again as a TStringList as well as deleting a Section or a key within a Section.

This component is used by other components in the set such as TAAFUndoRedo to store histories of actions on files while an application is not active. It is also very useful for storing user settings for an application.

3.11 RSLogFileManager (RSLogFileManager.pas) 

This non visual component manages a logging file. Again it allows you to capture the application settings to name the log file and store it in the application folder. If GetAppSettings is true, the filename will be created as ApplicationName.log. If it is not true then the entries in LogFileName and LogFilePath will be used.

The logging file can have a maximum number of lines set to control its size; when the size is exceeded the oldest lines are removed until the length falls with the limit. You can also set it to append lines to an already existing file or to overwrite it. You can also instruct the component to create a new file each day the application is started. If this option is set the old file will be saved as a .bak file.

You can add a line to the log file using the Add(Str: string) function and force it to write using the write function. The component does not allow you to specify a default value when a non existent key is first read but instead provides its own set of default values.

3.12 RSRegistryManager (RSRegistryManager.pas) 

This component simplifies using the Windows Registry. I'm not a great fan of the registry but it can be used if you want to make your application settings a bit harder to find or dangerous to tamper with. You can set the component to write to a give root key e.g. HKEY_CURRENT_USER and to a given sub key e.g. SimpleLogger. Within that sub key you can access name value pairs in boolean, DateTime, Date, Time, Float, integer and string values as well as Binary values passed in a Buffer.

3.13 RSSerialPortAccess (RSSerialAccess.pas) 

This non visual component provides access to a serial port. It can be used for controlling media equipment such as Videotape machines. The component can be set to access a given port number, provided it exists on your machine and you can use multiple components to access different ports. The baud rate property of the port can also be set.

The component can be either single threaded or use a separate thread for reading from the port. Usually when a port is written to, it will provide either a confirmation message, a data response or a failure message. The snag is that these responses can take a while to appear. If the component is single threaded, then your application will have to wait until the response has been received or until a ten millisecond timeout has expired. This should not happen because most equipment will not be happy to receive another message before it has acknowledged the last.

If the component has its threaded property set, a send call will return immediately. The response is then signalled by an OnDataReceive event which can be used to tell your code when to read from the component. If there is an outstanding message waiting to be sent or a returned value pending, the component's WriteData function will throw an exception which can be caught by your application code.

3.14 RSLogIn (RSLogin.pas) 

This visual component provides a log in dialogue for a user to enter a user name and password. The dialogue is invoked by calling execute. When the dialogue closes you can access its user name and password. You can then, for instance, look for them in a user database, showing a failure message if the combination was not found or allow the application to continue launching if it was.

3.15 RSGrid (RSGrid.pas) 

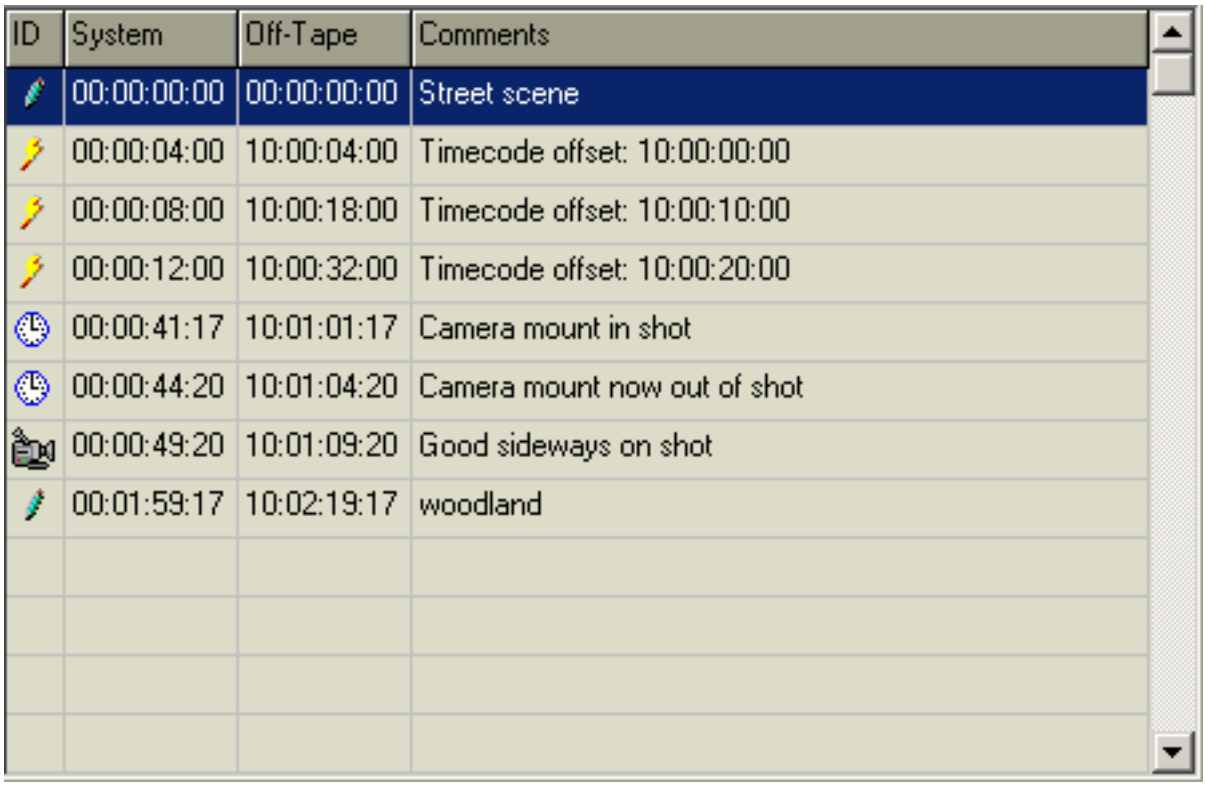
This visual component is derived from the standard TStringGrid. It provides the ability to load the column headings with a string list of captions, a configurable number of equal width columns to use for items such as timecodes and a number of other columns for data entries such as user comments. There is also an ID column at the left hand side that can be used to display an icon to identify the item type that the row belongs to. This component is useful for displaying user comments in an editor along with their time codes.

The icons are loaded into a row from a TImageList that holds the icons. The icons are loaded using a property Icon[Row: integer]. This is a read/write property and can be used to get the current icon or to change it. The value of the property corresponds to the index of the icon in the TIconList. You can use this value to identify a row for a given type even if all of the other rows have the same timecode and text entries.

The component also streamlines the addition and removal of rows from the grid. An InsertRow method allows you to insert a row at a given index, moving the contents of lower rows downwards. You can also remove a number of rows using the RemoveRow method which has a RowIndex and NumberOfRows parameters. When you remove rows, the contents of the lower rows are moved upwards to make a continuous list. This component provides all that you need for media applications without needing to use a custom component.

The component also provides some automatic behaviour when it is re-sized in that the final column re-sizes to fill the width and the number of lines is adjusted to either the number of none empty lines or the number of lines needed to fill the height. This avoids a messy looking grid.

The diagram below shows an example of how the component can be used. This is a grid showing shot comments, timecode breaks, timed comments and key frames, together with their system and off-tape timecodes.



4 AAF component descriptions

4.1 AAFCreator (AAFAccess.pas)

This non visual component allows you to create a new aaf file. It provides many low level routines that allow you to make files that describe just a set of media files, such as you might want after ingesting some content. You can also create a logging file that contains just a single clip that represents the ingested material before shot change detection, or files that contain only tape, film or import mobs to record information about the derivation tree of some media.

This component also contains the all important CloneExternal function that allows you to duplicate all of the derivation mobs from one file to another using just one function call. This call is truly powerful for providing the unbroken chain of derivation that professional users require when creating a new file or composition. It is the key to providing a list of media and the supporting metadata used in a composition.

The component has two classes called VideoFileProperties and AudioFileProperties in which you can store the data relating to files that does not change, basically anything other than the media length. This data is used for populating the media descriptors used in the file. The class can also include a set of legacy properties that may be required by some older applications.

All of the functions of this component are inherited by the other AAF Access components.

4.2 AAFLoggingAccess (AAFAccess.pas) 

This component adds the methods required to log a single piece of media represented in an aaf file. A logging file contains only one source MasterMob since it represents a single set of media files. It should also contain tape and file mobs if this information is available, and either an import mob if the media has been imported from a remote source or a recording mob if there is not physical source to be represented.

The logging file can be broken up into separate shots using either a shot change detector or manually and the component provides a means to insert these shots, remove them or adjust their positions. The component will look after the positioning of comments if there are any following a newly adjusted shot. These comments will be transferred into the new shot. Similarly, if a shot had a key frame that now follows an inserted shot it will be transferred to the new shot at the same position with respect to the media. These behaviours allow a file to be logged while a shot change detector is still processing it.

If a shot is removed, then its timed comments are moved to the previous shot at the correct times. If it had a shot comment, this is converted to a timed comment at the position where the removed shot started. If the position of a shot boundary is adjusted then the timed comments are transferred to the correct shot and the shot comments are retained.

These behaviours sound relatively simple but are in fact quite complex to implement. The reason is that the comments are held in separate MasterMobs, one for each shot. This structure simplifies the breaking up of an ingested shot into separate media items representing each shot. When this is done, the master mob and its derivation mobs already exist in the logging file and can be cloned into the new shot-related files.

The component provides a means to add, edit or delete a single shot comment for each clip and a set of timed comments. Each shot also has a key frame parameter that defaults to a zero position but can be positioned and assigned a user comment.

The component also provides a means to store the timecode break information that should have been captured when the media was ingested. This information is stored in the tape mob and provides the means to track a give video frame through the potential list of broken and wrong timecodes to arrive at the corresponding frame on a video tape, event when the video tape has a thoroughly broken timecode, as seems to be their habit.

The component also provides the means for linking from time positions to clips within the aaf file and for finding the list of comments attached to each clip. It can also provide the properties of the file media, the tape and film mobs and the import or recording descriptors. This information is essential for building a usable logging application.

The component is indexed for clips using a ShotID which is a one based index. An index of zero represents the MasterMob for the entire captured content. This requires some care when linking the component to other components such as an RSTimeline. This mapping is implemented in the AAFLogger component so you only need to worry about it when you implement your own code outside of this component.

4.3 AAFEditAccess 

This non-visual component extends the behaviour of the AAFLoggingAccess component to allow use of more than one source media set and adds the ability to specify video and audio effects and audio clip gains and levels.

It uses a different approach to commenting the file since it deals with a composition made up of numerous source media sets. Shot comments are attached to the composition SourceClips and refer to details of the composition itself not to the media. Timed comments are attached to the source MasterMobs and refer to details of the source information. The upshot of this is that shot comments are tied to the shots and will be lost when a shot is deleted whereas timed comments are tied to the media and will move as the timeline times are changed. The timings for timed comments are therefore specified with respect to the source media not the clip as it appears in the timeline. This puts some additional responsibilities on the handling of timed comments and means that graphical items such as comment readouts need to have their timings updated when the details of a timeline are changed. These behaviours are implemented in the AAFEditor component which should reduce the amount of developer effort required to make them work.

An editing file contains some rather complex structures to define the edits. The AAFEditAccess component hopefully avoids your need to understand these more than once, or even at all if the file can be read by another application. The AAFEditAccess is addressed in terms of ShotIDs which, for uniformity with the AAFLoggingAccess are one based. There are also some functions addressed in terms of ClipIndex which is zero based – apologies for this. These differences need to be born in mind when connecting it to components such as the RSTimeline which are indexed with zero based indices. Most of this translation is done for you in the AAFLogger and AAFEditor components so you only need to think about it if you implement your own code outside of these components.

The property pulling functions in this component have a ShotID parameter that allows them to access the file and derivation information of a particular shot. The entire contents of the file can be accessed by iterating through the clips and pulling each item out. Some clever code could be used to compile a list of whose content you have used and you could then use this to compile, say, an xml document that could feed a system to identify who you need to pay.

4.4 AAFProductIdent (AAFPack.pas) 

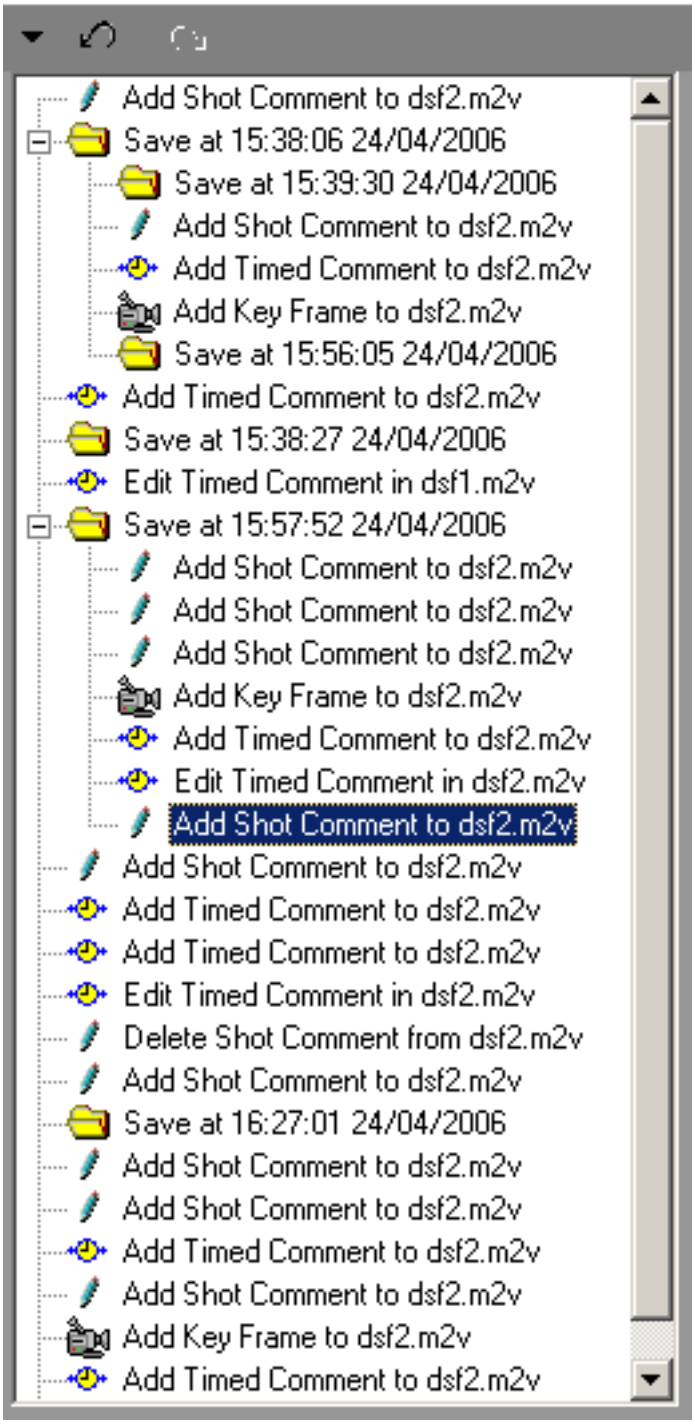
This component allows you to tell your AAF components the details of you application. Each component must have one of these attached to it and the same Product Ident can be attached to many access components. If you forget this, you will have lots of errors thrown up by the aaf implementation, in fact without it, almost nothing will work.

4.5 AAFRawStorageMemory,  AAFRawStorageDisk, 

AAFRawStorageCacheDisk 

These non visual components are meant for creating storage for storing media content. I have not had to use them since I have used DirectShow and standard files for my content. The general preference for professional broadcasters is to use media file formats for which a standard exists, or failing that, an operating system vendor provides components to handle.

The idea of wrapping media up inside a file is popular with equipment vendors but not with users since it normally means the media can be handled only by that vendor's applications. There has never to my knowledge, been a single vendor who provides a perfect application for everything a user might want to do, so the need for portability between applications is essential, and this means standardised media formats used for external media files. For this reason, my components do not use internally packaged media.



4.6 AAFUndoRedo

This visual component provides a means to record the history of operations on logging and editing files. This is a vital ingredient for professional media production which often involves making several different versions of a production using different combinations of media or timeline parameters. The component avoids the user's need to keep a whole string of editing projects with different names on the boil.

The component keeps its lists of project states in an Inifile stored under an ActionID that identifies the current project. The IniFile stores any number of different sections under different ActionIDs so the component allows a user to open different projects without loosing the recorded states of previous projects.

You can add items to the list using the AddItem Function, passing in a text description and an Image Index to point to an Image in a TImageList that will be displayed alongside the text description. This makes it easier for a user to identify what type of item each entry corresponds to. The component allows you to delete an entire ActionID section so that the IniFile does not get full of unused projects. This might be done for instance when the user deletes a tracked file.

There is also a parameter MaxLength that, if set to a non zero value, limits the number of items in the list to avoid big projects getting an unfeasibly large undo list. The list is flushed starting with the oldest entries until its length is less than the limit.

The component displays its list as a tree view with nodes representing points in the project history to which the user has back tracked and started again. The component supports moving back to these points simply by clicking them in the list and can be instructed to ask the user if that was really what was required. The list can also be navigated by stepping backwards and forwards using, for instance a pair of buttons. This will work while the following steps are unambiguous but if they represent a branch in the tree, the component will pop up its display to inform the reader that a choice is needed. Steps backward through the project history always succeed unless the user reaches the first entry.

The component has a set of Events that control the status of three control buttons, one to popup its display, and two for stepping forwards and backwards respectively. The diagram shows an example of a project history showing project saves and an assortment of comment events. The User has just selected the last item in the previous branch but has not yet made any new changes. When a change has been made, the other item branches will be collapsed to save display space.

4.7 AAFLogger 

This non-visual component houses all of the functions required for a logging application that are not related to the form design. This means that your TForm Units will contain only code that is related to the design of the form and simplifies making a new logging or an editing application that houses a logging facility.

The Instance will need to be linked to an AAFLoggingAccess, a TRSPlayerLIb for media replay, some Open and Close Dialogs for opening and closing logging files and an RSSafeDlg for closing files with suitable warnings. The component can also be assigned an TRSTimeline if the logging part of the application displays timeline information and an AAFUndoRedo component if the application requires it. The Component has methods that link to the non User Interface events on the AAFUndoRedo. It also provides a set of Methods that look like TActions which can be connected to an Action List for handling Insertion, position and title editing and deletion of shot changes; insertion, editing and deletion of shot and timed comments, and key frames. If you have additional operations then you can connect the TAction to a form event handler and implement your additional operations before or after calling the AAFLogger procedures.

Also included is handling of a TThreadCutDetector for adding shot changes to a freshly ingested file. This process again has methods that link to the events of the TThreadCutDetector to allow the updating of a timeline, and the assigned FLoggingAcces. This process assumes that there is a TRSTimeline attached to the AAFLogger and runs on a separate thread so the user can start logging while the media is being processed.

There is also a function that allows the shot titles to be 'corrected'. This function names all of the shots with titles starting with Shot 1 and ascending. This function can be used when the user had modified the decisions of the automated shot change detector and resets the shot names to something a little more uniform.

4.8 AAFEditor 

This non-visual component houses all of the functions required for an editing application that are not related to the form design. This means that your TForm Units will contain only code that is related to the design of the form and simplifies making a new editing application.

The Instance will need to be linked to an AAFEditAccess, an RSMediaSequence for media replay, some Open and Close Dialogs for opening and closing edit project files, an RSSafeDlg for closing files with suitable warnings and an RSTimeline. The component can also be assigned a AAFLoggingAccess if the application includes loading clips from a logging or a Bin file and an AAFUndoRedo component if the application requires it.

As with the AAFLogger, the Component has methods that link to the non User Interface events on the AAFUndoRedo. It also provides a set of Methods that look like TActions which can be connected to an Action List for handling title editing, Insertion, editing and deletion of shot and timed comments. If you have additional operations to perform then you can connect the TAction to a form event handler and implement your additional operations before or after calling the AAFEditor procedures.

It also has methods that allow appending, modifying and deleting audio and video transition effects; accessing audio clip levels and gains; cutting, copying, pasting and deleting clips in a timeline and setting video flip and flop properties. Many of these methods can be called directly by other component events as a result of user actions.

5 Mozilla component descriptions.

5.1 PJDropFiles 

These non visual components provide events which deliver the filenames of items dropped from Windows onto your form or a component on your form.

5.2 MRUList 

This non visual component provides a means of storing recently used files and displaying them in a popup menu assigned to the component. It stores its lists in the registry under HKEY_CURRENT_USER in the Software section under a Manufacturer Name and the Application Title. The entries go in a sub-key called Recent Files. You can also break this sub-key up into separate sections by using the AppSection property which puts the entries into a sub-key named from the value of AppSection. This means you can have more than one MRU List for different functions within an application.

The component tests to see if the file exists before loading into its popup list. What it does with an item that does not exist depends upon the code linked to the component's OnMissingItem event which contains a var parameter called DeleteIt. If this is returned as true, the component deletes the entry, otherwise it displays it as a disabled item in the popup menu so the user cannot select it.

The component also provides functions for getting the Selected File's directory and fully qualified filename, and to Clear, Load, Save and Refresh the list of items.

5.3 PersistentPosition 

This non visual component provides a means to store the form positions and sizes between application launches. It stores its entries in HKEY_CURRENT_USER in the Software section under a Manufacturer Name and the Application Title. It stores its values in a sub-key called Position. It stores the owner form's Left, Top, Height and Width properties, plus its form states. The component only has two specific property called Manufacturer and AppSection. Note that if you use more than one of these components on different forms, you must use different values of AppSection for each component.

5.4 RunOnce 

This non visual component checks to see if there is another instance of the application running. It does this by attempting to create a Mutex made from a wacky string of characters followed by the application name, if this fails then there is already an instance in existence. It provides Events for OnFirst instance which fires if there is no other instance, OnOtherInstance, which fires when another instance is fired up and allows its command line parameters to be captured by the first instance, and OnSecondInstance which fires when there is already another instance. If there is already another instance, the component focuses the previous instance's window and the OnSecondInstance event can be used to terminate the second instance.

6 Sourceforge location for the component packs

The components are all part of a single Delphi package called EditPackD6.dpk which compiles for Delphi 6. It will however compile for later versions although there are some modifications required to make one of its required libraries, DSPack2_31 compile. EditPack and the supporting libraries and demonstration applications can be found at http://www.SourceForge.net/projects/aaf-edit-pack. The only other components required are the mpeg decoding filters which can be bought from Elecard Main-Concept.

6.1 Supporting libraries

6.1.1 DSPack2_31

DirectShow compilation requires DSPack2_31 which can be found at http://www.SourceForge.net/projects/DSPack: This contains the JEDI DirectShow9 files and is required to compile some of the components in AAFEditPack. You can install it using DirectX9_DX.dpk in the packages directory. You will also need to add the src\DirectX9 folder to your Library path in Environment Options|Library.

6.1.2 AAF dlls:

The AAF Components require that the aaf dlls are present in your $Windows$\system32 folder. These can be found in the folder dlls. Copy AAFCOAPI.dll to the system32 folder and copy the entire aafext folder and its contents to your system32 folder. Alternately, you can download the sdk and compile them yourself.

You may also need to put MSVCP60D.DLL and MSVCRTD.DLL into your system folder if you do not have microsoft VC6+ installed and you are using the debug version of the aaf libraries.

6.1.3 DirectShow Filters:

If you want to use the media components in AAFEditPack then you will need to install some DirectShow filters. These are called RSAudioMixer.ax, RSVideoMixer.ax and CutDetect.ax You can put these anywhere on your machine but they are Com objects so they need to be registered. At the command prompt: type: regsvr32 $Full Directory$\RSVideoMixer.ax - etc and you should get a dialog saying the registration succeeded. If you move the files, you will need to re-register then at the new location. To remove the registration type: regsvr32 -u $Full Directory$\RSVideoMixer.ax

The source code of the DirectShow Filters is provided in the folder called DSFilters. There is a project group called DSFilters.bpg that loads them into Delphi. Note these filters use a modified version of BaseClasses.pas

There are also some test applications for these filters in the DSFilters folder. The editing test application has some hard-wired file locations for the Load buttons. These buttons allow you to quickly load a series of files with a single button push. You can change these to point to a folder on your machine.

6.1.4 Mpeg decoding filters:

You will also need to install either the Moonlight-Elecard player version 3.0 or higher, or Elecard/MainConcept filters version 4 or later to get access to their mpeg decoding filters. These have special indexing features required to get frame accurate replay and rapid scrub-play. The aaf sequence player currently replays m2v and wav files only and will not work without these filters. The filters are not open source and Moonlight are in liquidation but Elecard/MainConcept are going strong.

The moonlight filters are:

- mpeg2dmx.ax version 3.0 or higher - mpeg decoder
- mpgdec.ax version 2.0 or higher - mpeg demultiplexer
- mlcom.ax version 1.26 or higher - audio decoder
- mpeg2mux.ax version 2.2 or higher - for re-multiplexing streams
- dumppos.ax version 1.0 or higher for writing mpeg files

For frame accuracy you will also need:

- MLMIxReader.dll - index file reader
- MLMpgIndexator.dll - index file writer

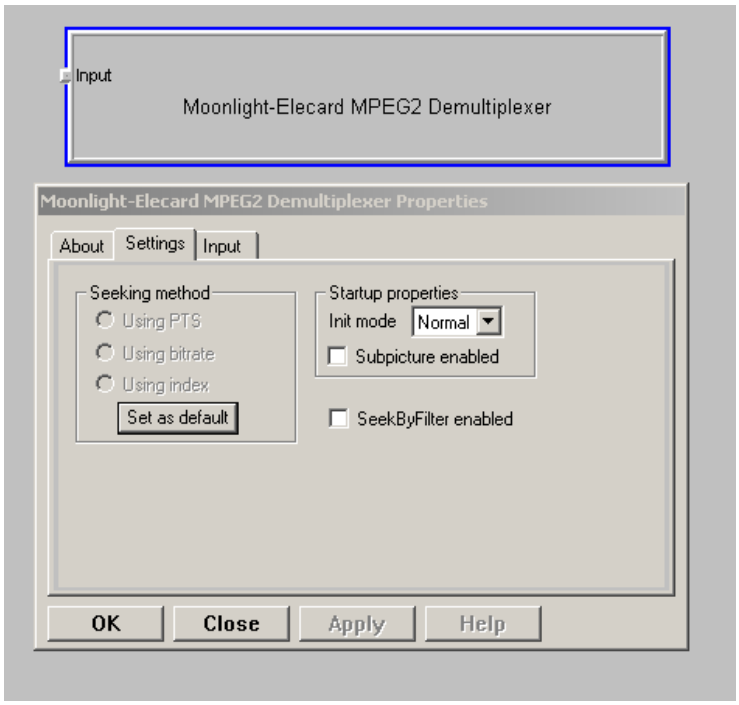
The Elecard/Main-Concept filters are:

- em2vd.ax - mpeg decoder
- empgdmx.ax - mpeg demultiplexer
- ELMux.ax - for re-multiplexing streams
- eleaudec.ax - audio decoder
- ESF.ax - for writing mpeg files

For frame accurate mpeg you will also need:

- ElMIxReader.dll - index file reader
- ElMpgIndexator.dll - index file writer

Some versions of the Moonlight mpeg de-multiplexer have a SeekByFilter property that defaults to true. This option cannot be set in code for all versions so it is essential to set it to false by selecting the filter using GraphEdit and setting the property to false in the filter's Settings page. If you do not do this, then a sequence of files in the MediaSequence component will not seek properly. To find this, load the filter using the Insert Filters command (Ctrl-F), open the DirectShow Filters item and Select Moonlight-Elecard MPEG2 Demultiplexer. Click Insert filter to add it to the graph. now right click the filter and click the properties entry in the popup menu. The properties dialog has a Settings page. If there is no SeekByFilter item then the version you have does not have this property and will work according to the Microsoft documentation.



There are no human readable references to the interfaces used by these components in the source code and tracing into the files provided in dcu form will not help. This is a condition of the sdk licenses. If you want them you will need to buy them - but they are well worth it.

The player components have an MpegCodecSet property. The parameters are

csDefault -> mpeg installed filters are used according to their priority.

csElecard2 -> uses Elecard player version 2 files without frame accuracy.

csMoonlight3 -> uses Moonlight player version 3 with frame accuracy.

csElecard4 -> uses Elecard4 filters with frame accuracy

There is also an open source set of filters from Gnu but these do not yet support frame accurate positioning.

You can now install AAFEditPack by clicking its package icon, then compiling and installing it. You should get a component tab called EditPack but before the components can be used, you will need to add the package folder to your Library path in Environment Options|Library.

The package contains mainly components written by me but also includes some Mozilla licensed components used in the demo applications. These are:

PJDropFiles from Peter Johnson.

PersistentPosition, MRUList and RunOnce from Colin Wilson.

If you already have these then exclude them from AAFEditPack

6.1.5 Demonstration applications

The demos folder gives some examples of code built using the EditPack components and some demonstrations of some of the components. The AAFDPT folder includes several projects that use the component pack. These are:

AAFDPT this is a video editor that is about 50% finished. It allows logging of an aaf bin file to include comments, shot change detection key frames etc. It also allows editing of content taken either from m2v and wav files or from an aaf file that wraps them and holds metadata. This application stores its projects and bin files in the aaf file format, which is the only publicly accessible file format for media content. This format is becoming increasingly used by professional application builder and offers many advantages such as the ability to get a record, including import source, tape and file information, for all of the content that you have used in your production. This is gold dust for professional users. There is a long todo list for this application that I can give to anyone who is interested in developing it.

Player is a media player that allows frame accurate positioning and rapid seeking through mpeg files, even if they are on a remote drive. This is really cool since without indexing in the player, seeking is very slow for longGoP variable bit-rate files. This application shows that mpeg is suitable for professional use and adds to its cost attractiveness. Other longGoP variable bit-rate formats would be equally good of course, as soon as someone makes an Indexer.

SimpleLogger is a logging application that allows you to take a matching m2v and wav file or a muliplexed mpg file captured by or imported into you machine. It is almost complete but needs some more testing. You can then do a shot change detection on it and document each of the shots using a comment per shot and timed comments within a shot. each shot can also have a key frame set. If your ingest method includes detection of off-tape timecode, you can also set some data that describes the continuity or otherwise of this timecode, which is crucial if someone else owns it and you need to pay royalties. The application opens and saves content as an aaf file and has a tool for creating a file from source media. The source media is always external since the aim behind aaf is to allow you to use anyone's production tool instead of being tied into one vendor, which will free the market up for all.

Reader and IndexBuilder are for reading and making index files for mpeg. The application for building an aaf file will make this index for you if it does not already exist but these tools are interesting for anyone who wants to learn about mpeg.

MakeAAFFile is a stand alone version of the tool from SimpleLogger.

Trimmer is only part completed and will allow you to separate out sections of an mpeg file whilst retaining the frame accuracy.