Interfacing the DivX Encoder (API)

Application Note

Release Date: August 25, 2005

Table of Contents

Document History

Document Version	Product Version	File Name	Release Date
1 Prerelease	DivX 6 CE SDK RC-1	an_divx6_interfacingencoderapi.pdf	04/15/05
1	DivX 6 CE SDK V3.0	an_divx6_interfacingencoderapi.pdf	06/17/05
1.1	DivX 6 CE SDK V3.0.1	an_divx6_interfacingencoderapi.pdf	08/25/05
1.2	DivX 6 for Linux	an_divx6_interfacingencoderapi.pdf	01/26/06

Document Change Log

Section Number	Page Number(s)	Description of Change
N/A	ii-1	Updated publication date and Document History
5	2.1.11	Modified "encode(..)" to "encodeFrame(….)".

1 Introduction

This application note describes the interface to the DivX encoder libraries (libdivx.so) and provides guidance in using them as part of the encoding DivX 6 video. This new version of the DivX encoder has changed to a C++ interface. This new API is intended to assist you in quickly and easily integrating the DivX Encoder into your application.

1.1 In This Document

This document contains the following sections:

- **Headers for the Encoder Interface**
  This section defines all the headers that define the encoder interface and provides guidance for setting them for your implementation.

- **Examples**
  This section describes several examples you may wish to follow.

2 Headers

The encoder interface is defined by the headers listed below. The subsections to follow provide information and guidance in setting these headers for your implementation.

- EncoderInterface.h
- Settings.h
- FormatInfo.h
- FrameInput.h
- FrameOutput.h
- DivXException.h

Optional:

- FrameResult.h
- Cli.h
- EncoderCallback.h
- FeedbackInterface.h

Because there are many settings that can potentially conflict (and invalid settings are configured to produce errors), we highly recommend that you catch and handle any thrown exceptions using a try/catch around all calls to the DivX encoder libraries; this enables you to locate any problems in configuring the DivX 6 Encoder. When you are configuring the Settings.h object, we also recommend that you catch Settings::Exceptions. For guidance in handling exceptions, refer to *Section 2.2.4* for Settings::Exception and *Section 2.10* for DivXException.

2.1 EncoderInterface.h

This header defines the primary interface to the encoder: it is able to create, maintain, use, and destroy a DivX encoder. This header is not the actual DivX encoder, but simply the usable interface: the actual class inherits from this class (and overloads the pure virtual functions to provide the real functionality). Since this class defines an interface, you cannot instantiate this object, however, you can create a pointer to the interface and use it to EncoderInterface* create(); an encoder.

Once you have determined which type of bitstream packing you prefer, call either bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) or bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult). To assist you in selecting your method, review the section below, which will help you understand the benefits and tradeoffs of each method for your particular application. Select the object that provides the output you require.

- ***Note:*** You should not mix these two calls within a particular instance of an encoder.

2.1.1 EncoderInterface* create();

This method creates an instance of an encoder. Unless you instantiate an encoder, none of the other functions have any meaning and results of any other function calls will be unpredictable at best. The EncoderInterface* create(); call needs to always be paired with a static void destroy (EncoderInterface* pIQEncoderInterface) in much the same way as you would pair new and delete (or malloc and free). This call will return a pointer to an encoder interface which the client must maintain; in the same way, you must not lose a pointer when you use new: you must keep a pointer to the newly created encoder (or you will create memory leaks because there is will be no way to destroy the encoder).

When you create an encoder, it is ready in a known good state, and no additional configuration is required. Depending on your source input and desired output, however, more configuration may be required. The DivX encoder is designed to be fully re-entrant so you can instantiate an array of orthogonal encoders, however, you must destroy them when you are done.

- ***Note:*** 7-8 internal frame buffers may be required for each instance, so consider your memory constraints carefully before creating multiple instances.

2.1.2 static void destroy (EncoderInterface* pIQEncoderInterface)

This method functions in the same way as using a delete call after new. When you are finished with an instance of an encoder, destroy it and reclaim all the memory.

2.1.3 Settings* getSettingsApi()

This method provides access to the settings object that allows a user to query and configure the encoder settings. For more details, refer to the *Encoder API Settings Functional Specification* included with this SDK, and to Section *2.2, Settings.h*.

2.1.4 void setFeedback(FeedbackInterface* pFeedbackInterface)

This method enables you to create a feedback interface that can give you real-time information during encoding. For an example, see the DivX VfW encoder feedback window (discussed in *Section 2.8, FeedbackInterface.h*).

2.1.5 void setEncoderCallback(EncoderCallback* pCallback)

This method enables you to get feedback from the encoder during configuration and a way to provide error messages to a client application. For further details see EncoderCallback.h section. If you do not pass in an EncoderCallback object, the encoder will not use the callback.

2.1.6 const char* getVersionInfo()

This method returns the version of the encoder library. The text will resemble something like "b1515-Feynman." This function will allow you to identify and discriminate versions in a client application.

2.1.7 void setFormatInput(const FormatInfo* pFormatInfo)

This method allows a client application to set the input format of the video being given to the encoder. This information includes colorspace and dimensions of the input video. The format information needs to be configured before actual encoding begins and once encoding has begun, this information cannot change. For further details, refer to *Section 2.3, FormatInfo.h*.

2.1.8 bool getFormatInput(FormatInfo* pFormatInfo)

This method allows a client application to verify the input format information that is called through the set method above; in principle, these should be identical. A return value of "True" indicates success, and "False" indicates a problem with the input format. For more information, refer to *Section 2.3, FormatInfo.h*.

2.1.9 bool getFormatOutput(FormatInfo* pFormatInfo)

This method retrieves the output format of the encoder. This information is largely controlled by the settings of the encoder. Returns "True" for success, while "False" indicates a problem with the input format or the settings (e.g. illegal resize under a profile). For further information, refer to *Section 2.3, FormatInfo.h.*

2.1.10 void deliverFrame(const FrameInput* pFrameInput)

This function delivers a raw, uncompressed frame to the DivX encoder. This raw frame should correspond to the input frame given above. This function does not encode the frame but simply gives the frame to the encoder to be later encoded by a call to either bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) or bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult). Call void deliverFrame(const FrameInput* pFrameInput)with pFrameInput->m_pFrameT = 0 to indicate the end-of-stream. For more information, refer to *Section 2.4, FrameInput.h.*

2.1.11 bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult)

This method compresses the delivered frame. It returns a frame in bitstream order and the caller must handle the packing. If no b-frames are enabled, the results should be the same as the bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) call below. However, with any GOV containing B-VOPs, you must be careful to pack the stream correctly.

For example:

$I_1 P_2 P_3 B_4 B_5 P_6 \ldots$

Must be ordered in the stream as follows:

$I_1 P_2 P_3 P_6 B_4 B_5 \ldots$

For proper decoding, the decoder must have P_6 before it can decode B_4 or B_5. Bi-directionally predicted frames predict from both the forward and backward P-frames around it, so both the prediction VOPs must be present in the decoder before you can proceed to the B-VOPs.

The proper syntax to call this function is as follows:

```
{
  bool frameEncoded = false;
  pEncoder->deliverFrame(&frameInput);
  frameEncoded  = pEncoder->encodeFrame(&frameOutput);
  yourPackBitstreamIntoContainerFunc(&frameOutput);
}
```

The pFrameOutput is a pointer to a structure that tells the encoder where it should write out the bitstream. This memory is always filled with a frame after a successful call to bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) The client is responsible for providing this memory to the encoder (to avoid excessive memory copies, for efficiency). The pFrameResult pointer is an optional parameter that the encoder will populate with information about the encoded frame. The sequenceNumber field of the structure must be used to know what frame the results refer to. For details, refer to the sections FrameOutput.h and FrameResult.h.

A return value of "True" indicates a successfully encoded frame; "False" indicates an error (typically, an error that the encoder is waiting for a frame to be delivered).

2.1.12 bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)

Unlike the previous method bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult), bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) can handle the bitstream packing for you. The technique required to use this function is somewhat more complex, however, since it handles the bitstream packing for you, the overall complexity to the client application is reduced. Internally, this method calls bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) to perform its job. The general form factor to follow is shown below:

```
{
  pEncoder->deliverFrame(&frameInput);
  while (!pEncoder->encode(&frameOutput))
  yourPackBitstreamIntoContainerFunc(&frameOutput);
}
```

A return value of "True" means that you should examine pFrameOutput (and possibly pFrameResult) to see what has been built. "False" indicates that a frame has been encoded, but a bitstream chunk cannot be produced yet, so you need to keep calling bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) until it goes true. Once bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) returns true, it is time to deliver the next frame.

- ***Note:*** After every I-VOP, the bool **encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)** will spit out a number of special placeholder frames that is equal to the maximum number of allowed B-VOPs. These frames are merely placeholders, and the resultant bitstream data is one byte in size (and is equal to 0x7F). This is required because of how the bitstream needs the frames to be ordered for proper decoding. After these placeholders, there is a one frame in, one bitstream chunk out, correspondence (but sometimes there are multiple VOPs in that chunk).

Unlike bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult), a call to encode is not guaranteed to produce a frame; one frame input does not imply one frame output. Because bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult) handles the bitstream packing, the pFrameOutput may contain zero, one, or two frames. The diagram below illustrates this process:

Deliver Frame		Encoder Decision		encodeFrame() return sequence	Bitstream output
1	→	I_1	→	true	I_1
2	→	P_2	→	true	0x7f
3	→	B_3	→	true	0x7f
4	→	B_4	→	true	P_2
5	→	P_5	→	false → false → true	P_5B_3
6	→	B_6	→	true	B_4
7	→	P_7	→	false → true	NotCoded (instruct decoder to display P_5)
8	→	B_8	→	true	P_7B_5
…					

The pFrameOutput is a pointer to a structure that tells the encoder where it should write out the bitstream. This memory is always filled with a frame after a successful call to bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) The client is responsible for providing this memory to the encoder (to avoid excessive memory copies, for efficiency). Refer to *Section 2.5, FrameOutput.h* for additional details

The pFrameResult pointer is an optional parameter that the encoder will populate with information about the encoded frame. We recommend initializing at least the sequenceNumber field to "-1" so that you know when this structure is filled by the encoder. Using this method, if the encoder fills the structure, it will get a number from 0 to the total number of frames, and you can remember the results accordingly. It is important to examine this structure on both the true and false returns of encode() (inside the while loop and after it) along with sequenceNumber to know the frame to which it refers. This call may report multiple results for a given frame. In the example above, the encoder returns multiple results for frame 3 because it first tries it as a P-VOP, then later as a B-VOP (after frame 5 is delivered). For this reason, you must use the last information for a given frame because the encoder can change the way it wants to encode a frame. For details on the structure, refer to *Section 2.6, FrameResult.h*.

2.2 Settings.h

The Settings class enables you to configure the encoder. To get started in using this header, perform the following preliminary steps.

Step 1. Enclose any attempts to change settings with a try/catch (Settings::Exception). The Settings class is designed to throw an exception when an attempt to perform illegal operations is made [e.g. getType("this_parameter_does_not_exist"]. This will allow you to quickly isolate any problems in configuring our encoder.

Step 2. Access the settings via the getSettingsAPI() function in the encoder interface.

Step 3. Check that your header file and library versions are in sync so you can call isCorrectHeader() to verify this.

Step 4. Make sure that "working_folder" is set properly so the encoder can function properly. You must tell the encoder a safe place to write persistent file output. Because this cannot be set internally by the codec, it is the only parameter that is absolutely required before encoding (depending on your application other parameters may also need to be changed, but that is application-dependent).

Step 5. Perform any customization tasks to configure the encoder for your implementation, referring to the *Encoder API Settings Functional Specification* included with this SDK and to the subsequent topics in this section:

2.2.1 Parameter Names

2.2.2, Parameter Types

2.2.3, Setting and Querying Parameters

2.2.4, Exception

2.2.5, Copying and Saving the Settings Structure State

Step 6. Lastly, consider the conversion of the fourCC to an "encoder" enumeration:

```
static Enum fourCC2Encoder(FourCC fourCC).
```

Currently, the encoder only supports MPEG-4, so the other enums will be rejected. Future releases of the encoder may support other fourCC codes so that this function can provide an easy means of conversion between an externally recognized fourCC code and the correct DivX encoder to create it.

2.2.1 Parameter Names

Every parameter is addressable by its name. For a complete list of parameter names, refer to the *Encoder API Settings Functional Specification* included with this SDK. The Settings class provides the functionality to discover the available settings during runtime.

To discover the name, you can query the settings structure using the getName() function, as shown below.

```
//assumes an pEncoder is already created
int i = 0;
Settings::Name next;
const Settings* pSettings = pEncoder->getSettingsApi();

for(next=pSettings->getName(i); next.isValid(); i++)
{
  yourAddNextParameterFunction( next );
  next = pSettings->getName(i);
}
```

You can address a parameter by either using the Name object found in the settings class, or by using string literals. Examples of each are shown below.

```
Settings::Name mode("rcmode");  //create the Name object
pSettings->getType(mode);       //pass into the settings
```

OR

```
pSettings->getType("rcmode"); //in this case the string will be
                              //automatically cast into a Name object
```

2.2.2 Parameter Types

Once you query a parameter's name, the next step is to determine what type of parameter it is. Each parameter has a type and you can only set and query it if you know its type. To learn the type of a parameter, call getType() (using either a Name object or string literal):

Listed below are the possible parameter types:

- BOOLEAN (true = 1 or false = 0)
- INTEGER (an int)
- DOUBLE (a floating point with double precision)
- ENUM (an enumerated parameter, list of enumerations at the bottom of the header)
- STRING (an ASCII string parameter, char*)
- DATA (binary data)

> ***Note:*** getType() does not return the actual type; it simply returns the enumerated types from the Settings object. This tells you how to set and query the parameter (discussed in the next section).

As you will see in the next section, every parameter requires that you set or query it according to its type.

2.2.3 Setting and Querying Parameters

Once you know a parameter's name and type, the next step is to set and query the parameter. The topics in this section help you understand the following aspects of this task:

- Parameter Precedence
- Parameter States
- Boolean Types
- Integer Types
- Double Types
- Enumerated Types
- String Types
- Binary Data Types

Note: When you perform "illegal" operations on the Settings, the standard response is to throw an assertion (e.g. set/query parameters that do not exist, do not use the typed method that matches the type of the parameter, etc).

2.2.3.1 Parameter Precedence

The first step in setting and querying a parameter is to understand its precedence. Precedence tells you what parameters are allowed to affect other parameters. This is important because certain parameters of the DivX encoder can and will change other parameters in a cascading effect. For example, if you set the "profile" parameter, there are a large number of dependent parameters including, VBV values, output resolution, frame rate, and many others. So when you set a parameter in the encoder, it is able to propagate that change throughout all the encoder settings Settings. It allows a client application to have less responsibility and knowledge of the inner workings of the DivX encoder.

To query the query a parameters precedence call getPrecedence(). Return values range from 0, the highest precedence, down to minus (-) the total number of parameters [the precedence is simply -n in context of the getName(n) call]. A setting is allowed to (but doesn't always) affect settings with a lower precedence. For example, the "profile" parameter is the second highest parameter in precedence (=-1), and it is allowed to change basically every setting when you change the profile. Conversely, if you were to change the "resize_mode", it could not change the "profile" because it has a lower precedence.

In general, it is best to set parameters in descending order of precedence to ensure that all your settings are maintained.

2.2.3.2 Parameter States

A few states are associated with each parameter in the Settings object. They all return boolean values to indicate if that particular state applies to a given parameter. These states are discussed below.

- **bool isReadOnly(Name name) const**
  This parameter cannot be set by the client application. There are any number of reasons this can be true.

- **bool isEnabled(Name name) const**
  Consider the "max_b_frames" setting. When this parameter is set to zero, no B-VOPS will be inserted into the bitstream. You may, therefore, have the number of B-VOPs set to one, but then set the "profile" to handheld. This profile does not allow B-VOPs, so the encoder remembers that you want one B-frame, but it disables (isEnabled()=false) the "max_b_frames" setting. This disables the B-VOPs while still remembering your state (so when you return to a profile that allows B-VOPS, you do not have to reset the "max_b_frames"). The function showDisabledSettingsInNextCliRender() allows you to show disabled settings when you render a CLI; otherwise, the CLI will only show non-default, enabled parameters.

- **bool isDefault(Name name) const**
- **virtual void resetDefaults()**
- **void makeCurrentSettingsDefault()**
  These three functions allow you to query and set the defaults of all the encoder parameters. When you create an encoder, all the parameters (except "working_folder") are initialized to a safe configuration. Every time you EncoderInterface* create(); an encoder, it always comes up in this same state. The isDefault() function allows you to find out if the parameter is in its default state. The makeCurrentSettingsDefault() method allows you to change the defaults to whatever is currently in the Settings object, however, this is not persistent across EncoderInterface* create();/static void destroy (EncoderInterface* pIQEncoderInterface) calls (EncoderInterface* create(); always creates an encoder in the same state, there is no way to alter this).

2.2.3.3 Boolean Types

The following two functions are used to query and set Boolean types:

- void setBool(Name name, bool value)

- bool getBool(Name name) const

  Note: If you pass in a name that is not a Boolean type variable, or try to set a read-only, an assertion will be thrown.

2.2.3.4 Integer Types

The following four integer-based functions are used to control the parameters:

- void setInt(Name name, int value)

- int getInt(Name name) const

- int getIntMin(Name name) const
- int getIntMax(Name name) const

The getIntMin() and getIntMax() allow you to query the minimum and maximum values of a **specific** parameter. Only integer type parameters can be addressed with these functions (other types will throw assertions).

2.2.3.5 Double Types

Double types are very similar to integer types, and are listed below:

- void setDouble(Name name, double value)
- double getDouble(Name name) const
- double getDoubleMin(Name name) const
- double getDoubleMax(Name name) const

Much the like the integer types, the min and max functions allow you to query the limits of a **specific** parameter. If you address other types of parameters with the double functions, an assertion will be thrown.

2.2.3.6 Enumerated Types

The enumerated types use the following functions:

- void setEnum(Name name, Enum value)
- Enum getEnum(Name name) const
- Enum getEnumMask(Name name) const

In order to use enumerated types, it is important to understand how the enumerations are defined. You will find the enumerated #defines at the bottom of the Settings.h file. You will also notice that the types are defined by bit shifting. You can set and query the enumerated types taking into consideration the #defines. The last function allows you to query the allowable enumerations of a **specific** parameter.

For example, three options may be defined as:

```
#define EXAMPLE_FIRST 1<<0
#define EXAMPLE_SECOND 1<<1
#define EXAMPLE_THIRD 1<<2
```

In this example, the "example" parameter (not a real parameter) is queried for its enum mask, and 0x05 = 5 = EXAMPLE_FIRST | EXAMPLE_THIRD is returned. This would mean that the second option, EXAMPLE_SECOND is not currently available. This is similar to querying the limits of an integer or double type. When the CLI with enumerated types is used, furthermore, the value passed is the number of bit shifts. In order to set the example parameter to the third option, therefore, the CLI syntax might look like this:

```
-example=2
```

For a more detailed description of the CLI, refer to the *Encoder API Settings Functional Specification* included with this SDK.

> ***Note:*** As with all the other set/query methods, a type mismatch will throw an assertion.

2.2.3.7 String Types

Some parameters carry string values, which you can set and query using the following two functions:

- void setStr(Name name, const char* value)
- const char* getStr(Name name) const

You do not need to allocate any memory to hold the queried strings: you are provided with a const pointer to the actual buffer. When you use the set function, you can either use string literals ("string literal"), or you can allocate a string buffer and then pass the pointer in. The parameter will make its own copy of the data, so you do not need to allocate any memory you pass in.

> ***Note:*** You cannot change the default values of strings because the parameters are declared with default values that are string literals and the space is limited; instead using of complicated logic to manage this memory, the operation simply isn't allowed.

2.2.3.8Binary Data Types

No Binary Data Type parameters are currently available, however, the API is present to support the functionality for future requirements.

- void setData(Name name, const uint8_t* buff, int len)
- const uint8_t* getData(Name name) const
- int getDataLen(Name name) const

Since there are currently no binary data type parameters, calling these functions will only result in an exception.

2.2.4 Exception

The Settings object defines its own exception so you can handle any exceptions caused by improper use of the settings structure. You should always place any settings calls within a try/catch block. To prevent unhandled exceptions, you need catch these exceptions and then use the information to correct any errors within your program. Any calls that are not expressly permitted will throw an exception, as shown below.

```
// C++ exception type used by the settings API.
class Exception
{
public:
  Exception(const char* info);
  const char* getErrorMessage() const;
private:
  const char* info;
};
```

Your code, therefore, should look similar to the example below:

```
try
{
  //assuming pEncoder points to a valid, create()'ed encoder…
  Settings* pSettings = pEncoder->getSettingsApi();
  pSettings->getType("nonsense");
}
catch(Settings::Exception& ex)
{
  YourPrintToConsoleFunction( ex.getErrorMessage );
}

...

//Would generate the following (because "nonsense" doesn't exist)

invalid name
```

If you follow this model, you can ensure that any improper use of the Settings will not cause undesired results in your application.

2.2.5 Copying and Saving the Settings Structure State

A copy constructor is provided for the settings structure. Additionally, the = operator is overloaded to allow the state to be easily copied. Combining this functionality allows a client application to easily query, save, and restore the state of the settings. There are a number of ways this can be useful to a client application: for example if you have a GUI that configures the codec, it is easy to do the following:

```
EncoderInterface* myEncoder = 0;
myEncoder = EncoderInterface::create();

Settings* savedSettings(myEncoder->getSettingsAPI());

//change settings in the gui using the getSettingsAPI(), but
// user decides to cancel

*myEncoder->getSettingsAPI() = *savedSettings;
```

You can also use the CLI to set and store the state;for full details, refer to *Section 2.7, Cli.h.*.

2.3 FormatInfo.h

This structure describes a compressed or uncompressed video format; the elements of the structure are listed below:

- **FourCC fourCC**: fourCC of the video format.
- **int bpp**: bits per pixel (zero if not known). This field is used to distinguish the various uncompressed RGB formats.
- **int width**: image width in pixels.
- **int height**: image height in pixels.
- **int inverted**: Set non-zero if the bottom line of the image appears first in the buffer.
- **int pixelAspectX**: horizontal part of the pixel aspect ratio.

- **int pixelAspectY**: vertical part of the pixel aspect ratio.
- **int sizeMax**: maximum size (in bytes) of a video frame in this format.
- **int timescale**: number of units of time in a second (e.g. 1000 implies milliseconds)
- **int framePeriod**: frame duration in units of timescale (e.g. if timescale = 1000, 25 fps means framePeriod = 40). In the case of a variable frame rate, this should be set to the maximum expected frame period.
- **int framePeriodIsConstant**: should be set to "1" if frame rate is constant, otherwise "0".

2.3.1 int FormatInfo_getTotalPixels(const FormatInfo* pFormatInfo)

This function is included for convenience. It multiples the height and width of the the formatInfo structure to determine a number of pixels.

2.3.2 int FormatInfo_getFrameSize(const FormatInfo* pFormatInfo)

This function is included for convenience. It multiplies the height and width of the formatInfo structure and multiplies to determine a number of pixels, and then multiplies by the bits-per-pixel and divides by 8 to determine bytes. This only functions when bits-per-pixel are known and properly set.

2.3.3 double FormatInfo_getFramerate(const FormatInfo* pFormatInfo)

This function is included for convenience. Using the formatInfo structure, this function returns the timescale divided by the frame period to yield frames-per-second (fps).

2.4 FrameInput.h

This structure has 3 members:

- unsigned char* imageT
- unsigned char* imageB
- int timestampDisplay

This allows the user to point at the even lines ("T"op field) and odd lines ("B"ottom field) individually. With progressive input, both pointers should be equal. The timestampDisplay is simply the timestamp of the frame in units of timescale (see formatInfo structure).

2.5 FrameOutput.h

This object is simply a container for the output of the encoder. All the members are public variables, and are listed below.

2.5.1 unsigned char* bitstreamBuffer

This is a pointer to a buffer that is owned by the client application. This memory is not managed by the encoder, it is simply where the encoded bitstream is stored. The encoder uses this memory to avoid excessive memory copies. See the next section for comments on size.

2.5.2 int sizeBitstreamBuffer

This member tells the encoder how much memory is available for bitstream writing. This is an input and should represent the memory allocated for the bitstreamBuffer. There is no absolute rule for the size required by the encoder, but we recommend that you provide 12*height*width in bytes. This will usually be excessive, but the alternative is to risk undefined behavior as the encoder writes off the end of the bitstream buffer. Error checking is removed from release builds because it tremendously slows the encoder.

The actual output from the encoder is very content-dependent and will usually be nowhere near this limit, however, the theoretical limit is near the recommended number above. We do not recommend economizing on the memory in this area.

2.5.3 int sizeBitstream

This object represents the actual size of the bitstream written out to the bitstreamBuffer. This field is populated by the encoder.

2.5.4 bool keyframe

This object is set to "True" if the output bitstream describes a keyframe.

2.6 FrameResult.h (optional)

This is an optional parameter for the encoder. If you wish to learn more about the FrameOutput from the encoder than the FrameOutput size and if it is a keyframe, you should use this object. The elements of this structure are listed below. Moreover, you must consider the sequenceNumber. You cannot assume that the frame you delivered is the one that is described by this structure. You may also get multiple frame results that refer to a single frame, you must use the latest one to make sure you have the correct information.

- **int bitcountMotion**: number of bits of encoded frame used for describing motion.
- **int bitcountTexture**: number of bits of encoded frame used for describing texture.
- **int bitcountStuffing**: number of bits of encoded frame used by stuffing.
- **int bitcountTotal**: total number of bits in encoded frame.
- **int quantizer**: actual frame-level quantizer used to encode frame.
- **int sequenceNumber**: display-order sequence number of encoded frame.
- **int motionVectorSum**: sum of logarithms of motion vector magnitudes.
- **int motionVectorCount**: count of motion vectors used to generate motionVectorSum.
- **char frameType**: actual frame-type ('I'/'B'/'P') of encoded frame. "0" = no frame produced. Packed B-VOPs will be labeled as "B".
- **int timestampDisplay**: display/composition timestamp of encoded frame.
- **float psnr**: PSNR of encoded frame. If "0" or negative, PSNR is unavailable.

2.7 Cli.h (optional)

This header defines a namespace that allows you to use the DivX codec's Command Line Interface (CLI) to set and read settings. In addition to the defined functions the standard C++, insertion and extraction operators are overloaded to insert and extract CLI strings onto and from the Settings object. Unlike accessing the Settings directly via the API, the CLI is much more forgiving when it receives unknown input: it typically ignores unknown input instead of throwing exceptions. The elements of this structure are listed below.

2.7.1 void render(std::string& cli, const Settings& settings) / <<

This is a query method. This function defines how to take the current encoder settings and render a CLI that represents them. This function and the stream extraction operator (<<) function in the same fashion.

2.7.2 void parse(std::string& cli, const Settings& settings) / >>

This is a set method. This attempts to parse a CLI and attempts and apply it to the Settings. The parse() function and the stream insertion operator (>>) function in the same fashion.

2.7.3 void showDisabledSettingsInNextCliRender(void)

This function allows you to render any disabled settings for the next call to render() or the extraction stream operator (<<). For example, if B-VOPs are disabled because of the profile, it will still tell you the current setting. If you do not call this on a render() or extraction, the "max_b_frame" parameter will be dropped from the CLI string because it is disabled. Therefore, the "max_b_frame" would be returned to it's default value (currently 0) if and when this rendered CLI is parsed() or inserted (>>) into the encoder's Settings.

2.7.4 static void base64Encode(const uint8_t* buffer, int size, char* string)

Encodes a binary data block into a base-64 textual representation.

2.7.5 static void base64Decode(const char* string, uint8_t* buffer, int* size)

Decodes a base-64-encoded string into a binary data block.

2.8 FeedbackInterface.h (optional)

The feedback interface is an extremely powerful tool that allows a client application to get and provide real-time feedback during encoding. If the feedback interface is enabled, the encoder will make these function calls during runtime. The client application is responsible for inheriting from this class and further defining how the functions are implemented. By default, the functions provide empty implementations so that if the class is not used, the functions should be optimized out (or at least provide very little impediment to the encoder speed). The feedback information is controlled by the "enable_feedback" Setting (a Boolean type).

All the functions listed below will block the encoder. For example, if the notifyEndFrame() function pops up a message box, the encoder will stop and wait until the user clicks "OK" until it proceeds [in fact, the notifyEndFrame() function is used by the DivX VfW feedback window to provide pause and the frame advance functions]. The feedback window can slow encoding if it is not done in a quick and lightweight manner. The VfW feedback window is an excellent illustration of the power of the feedback interface. It is up to you to maintain and use any or all of the information provided in any manner you see fit. The feedback interface is extremely flexible and you can use any of the parts for your application. Despite this, the FeedbackInterface is entirely optional. If you do not wish obtain any of the following information, simply ignore it and set "enable_feedback" to "False" (the default is true).

> **Warning:** *When using a threaded GUI to service the feedback information, beware of the dangers of shared information. The encoder may be updating information while the GUI is attempting to read the information. Take the time to set up the proper semaphore/critical section/mutex to guard against any threading issues.*

The methods of the object are are listed below.

2.8.1 void setEncodingDouble(const char* type, double value)

These parameters are valid across an entire encode. The types included are listed below:

- "frame rate": the encoded frame rate
- "pass": the pass of the encode

2.8.2 void setDimensions(int width, int height)

This callback provides the feedback interface with the dimensions of the encoded image. This is also an encoding scope parameter.

2.8.3 void setFrameDouble(const char* type, double value)

These parameters can change from frame to frame. The frame-level feedback parameters are listed below:

- "quant": the frame quantizer
- "PSNR": the frame PSNR
- "gmc_block_frequency": The percentage of GMC macroblocks

- "texture": the bits used to code the textures of the frame (RLE DCT data)
- "motion": the bits used to code the motion vector data
- "bits_1v": the total number of bits used for single motion vectors
- "bits_4v": the total number of bits used for 4mv motion vectors
- "part_4v": the percentage of total motion bits used for 4mv motion vectors
- "GMC": Boolean value for to indicate S-VOP
- "bits": total bits used to encode a frame

2.8.4 void setMacroblockDouble(const char* type, int x, int y, double value)

This function provides information about individual macroblocks (16x16 pixels in the luma plane). The (x,y) coordinates give the position of the macroblock for which information is being recorded and (0,0) is the upper-left corner of the image. The value is the double value that describes the type of information that is contained at macroblock (x,y). The units of (x,y) are macroblocks.

The types of data are listed below:

- "dct_type": the DCT type used for the macroblock
- "quantizer": the quantizer used for the macroblock
- "bits": the bits used to code the block
- "sad": sum of absolute differences (encoded versus source)
- "brightness": the average brightness of the luma
- "deviation": the deviation from the "brightness" level (like SAD, but with brightness)
- "4mv": A Boolean for 4mv
- "qpel": (mv_x & 3) + 4*(mv_y & 3)
- "field_mode": field_prediction ? ((forward_top_field_reference?1:0)+2*(forward_bottom_field_reference?1:0)) : 4)
- "GMC": mcsel
- "mb_type": the MPEG-4 defined macroblock type
- "maxdev": the difference between the frame-level quantizer and the macroblock quantizer
- "PV level": the psychovisual level
- "mv_x": the x component of a macroblock with one motion vector
- "mv_y": the y component of a macroblock with one motion vector
- "mv2_x1": the x component of the backward motion vector of a bi-directional macroblock
- "mv2_y1": the y component of the backward motion vector of a bi-directional macroblock
- "mv2_x2": the x component of the forward motion vector of a bi-directional macroblock
- "mv2_y2": the y component of the forward motion vector of a bi-directional macroblock
- "mv4_x1": the x component of block 1 motion vector of a 4mv macroblock

- "mv4_y1": the x component of block 1 motion vector of a 4mv macroblock
- "mv4_x2": the x component of block 2 motion vector of a 4mv macroblock
- "mv4_y2": the y component of block 2 motion vector of a 4mv macroblock
- "mv4_x3": the x component of block 3 motion vector of a 4mv macroblock
- "mv4_y3": the y component of block 3 motion vector of a 4mv macroblock
- "mv4_x4": the x component of block 4 motion vector of a 4mv macroblock
- "mv4_y4": the y component of block 4 motion vector of a 4mv macroblock

2.8.5 void setFramePointerType(int index, const char* name)

This function is intended to be used in conjunction with setFramePointers(). This is an encode scope call—in other words, it does not change during an encode. This simply informs the user of the feedback interface what types of images to expect from the setFramePointers() call. Current image types are listed below:

- "encoded frame": a picture of the encoded frame, index=1;
- "difference": the residual textures, index=2
- "compensated frame": output from motion compensation, index=3

These images can be understood as follows:
"encoded frame" = "difference" + "compensated frame".

This function will be called once per encode per type (i.e. it will be called three times at the start of an encode to tell you that you can expect the three image types above; prepare yourself to receive them.). This function will be called before setFramePointers()

2.8.6 void setFramePointers

(const char* name, const unsigned char* pY, const unsigned char* pU, const unsigned char* pV, int iStrideY, int iStrideUV)

As mentioned in the previous section, the setFramePointerType() call will tell you the type of images to expect during encoding. The next step is to provide you with the pointers to the actual images. This function accesses the encoder's internal frame buffers (hence the const qualifier on the pointers).

> ***Warning:*** *Do not attempt to modify these buffers. The frame buffers are read-only and they are for informational purposes.*

Neither the "compensated frame" pointer nor the "difference" pointer change over an encode, however, the "encoded frame" pointer varies with frame type (I, P, or B VOP) so you must be able to follow this change. Unless you inform the encoder of which images you require (with the appropriate index in getActiveImage()), the image will not be built.

2.8.7 int getActiveImage()

This is an important function to use in conjunction with the index from void setFramePointerType(int index, const char* name). The encoder informs the client application of what images are available to it and the index associated with those images. The client application is then responsible for pushing the current active image back down to the encoder so

that it knows what images it needs to build for the feedback interface. Conceptually, this function is the encoder asking the client application which image to build; you can only build one image at a time. After you deliver the next frame, you can change the image type. See void setFramePointerType(int index, const char* name) (previously discussed) for the indices. If you do not require any images, simply return the default value of "-1".

2.8.8 void notifyBeginFrame(int frame)

This method informs the feedback that it is attempting to encode the n^{th} "frame". This functional may make multiple passes at this frame; it may pass multiple times with the same frame number.

2.8.9 bool notifyEndFrame(int)

This function is called when the encoder has finished encoding a frame. The integer parameter is a char that represents the frame type. When cast to a char, it should return "?", "I", "P", or "B"'. The "?" signifies re-encode, and the others signify the encoded frame types.

The Boolean return value of this function provides a great deal of power to the feedback interface because the feedback interface can chose to accept the frame with a "True" statement or reject the frame with a "False" statement. If the frame is accepted by the feedback interface, the encoder moves on, if not, it will continue encoding the frame until this value is set to true. (Meanwhile, you can attempt to change Settings to get the results you desire.)

2.8.10 void print(int level, const char* fmt, ...)

This method provides textual debug information to the feedback object. It is like a printf() with a an ability to control the level of detail of the messages. The levels are as follows:

- 0 = no tracing
- 1 = codec-level tracing
- 2 = frame-level tracing
- 3 = slice-level tracing
- 4 = macroblock-level tracing
- 5 = bitstream-level tracing

2.9 EncoderCallback.h (optional)

The encoder callback allows the encoder to call back to the client application with various status updates including errors. The messages that the encoder will produce are listed in the header file. The strings are provided so that a translation layer may be inserted if desired/required. These functions must be overloaded in the client application. The elements of this structure are listed below.

2.9.1 int getVersion()

For future use.

2.9.2 void enable(bool enable)

This is a passthrough for the "use_dialogs" Setting. If this is not enabled, you should not show any dialog boxes.

2.9.3 bool promptYesNo(const char* caption, const char* msg, bool bdefault)

For future use. This function will allow the encoder to attempt to prompt a user for a "yes" (return "True") or "no" (return "False") answer. It allows both a message and a caption for the prompt. It also informs the client of the "default" answer.

2.9.4 void errorMessage(const char* caption, const char* msg)

This allows the encoder to send an error message to the client application.

2.9.5 void setProgress(bool enable, int percent, const char* caption, const char* msg)

At various points in the encode process, the encoder may be "thinking". This callback allows the client application to determine what the encoder is doing and the encoder's progress. Tthe message and caption are provided to the client.

2.10 DivXException.h

All calls to the DivX encoder need to be inside a try/catch block. This structure includes the following functions:

- int GetValue() const
- const char* GetFile() const
- int GetLine() const

and for WIN32 specific applications:

- int GetExceptCode() const
- struct _EXCEPTION_POINTERS* GetExceptInformation() const

In general, the exceptions are provided for informational purposes. This information may be helpful in the event that you need to contact DivX, Inc. to solve a problem.

For example:

```
try
{
  EncoderInterface* pEncoder = NULL;

  pEncoder = EncoderInterface::create();

  //do your thing...

  EncoderInterface::destroy( pEncoder );

}
catch(DivxException& ex)
{
  //something went wrong
  ofstream errorLog("c:\\divxError.txt")

  errorLog << "value = " << ex.GetValue() << endl;
  errorLog << "file = " << ex.GetFile() << endl;
  errorLog << "line = " << ex.GetLine() << endl;
}
```

Potential exception values are listed below:

- ENC_BUFFER = -2: Returned by encore() when bitstream is too long for the buffer provided by caller. Only returned by special builds of the codec that test for buffer overflow.

- ENC_FAIL = -1: Returned by bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)/bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) when call failed

- ENC_OK = 0: Returned by bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)/bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) when call succeeded

- ENC_MEMORY = 1: Returned by bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)/bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) when encoder has failed to allocate sufficient memory.

- ENC_BAD_FORMAT = 2: Returned by bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)/bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) if input video format proposed by user is not supported.

- ENC_INTERNAL = 3: Returned by bool encode(FrameOutput* pFrameOutput, FrameResult* pFrameResult)/bool encodeFrame(FrameOutput* pFrameOutput, FrameResult* pFrameResult) if there was an internal problem.

3 Examples

This section provides a few examples of how to use the API. It is the user's responsibility to determine how to best use the DivX Encoder Libraries (via your client application). These examples are intended to be a starting point to your implementation. Depending on your application, you may have additional requirements.

> ***Note:*** Functions that begin with "your….", such as "YourGetInputFrame", indicate that these functions must be provided by your client application. This code has not been compiled and may not be syntactically correct.

This section provides a mixture of code snippets that is intended to get you started. It is designed to show you the general form and function of the DivX codec libraries.

3.1.1 Creating an Encoder

```
try
{
  //create a pointer to the interface
  EncoderInterface* myEncoder = 0;

  //create an encoder
  myEncoder = EncoderInterface::create();

  //make sure the encoder is instantiated
  assert( myEncoder );

  //use encoder

  //clean up when you no longer need the encoder
  EncoderInterface::destroy( myEncoder );
}
catch(DivxException& ex)
{
  //something went wrong
  YourHandleDivXExceptionFunction( ex );
}
```

3.1.2 Creating Multiple Encoders

```
try
{
  //How many
  const int numEncoders = 3;
  int i = 0;

  //create a pointer to the interface
  EncoderInterface* myEncoders[numEncoders] = {0};

  //create an encoder
  for( i = 0; i<numEncoders; i++ )
  {
    myEncoders[i] = EncoderInterface::create();

    //make sure the encoder is instantiated
    assert( myEncoder[i] );
```

```
  }

  //use encoders

  //clean up when you no longer need the encoders
  for( i = 0; i<numEncoders; i++ )
  {
    EncoderInterface::destroy( myEncoders[i] );
  }
}
catch(DivxException& ex)
{
  //something went wrong
  YourHandleDivXExceptionFunction( ex );
}
```

3.1.3 Create and Configure an Encoder

Note the ability to use the Name object or a string literal when addressing the settings.

```
try
{
  //create a pointer to the interface
  EncoderInterface* myEncoder = 0;

  //create an encoder
  myEncoder = EncoderInterface::create();

  //make sure the encoder is instantiated
  assert( myEncoder );

  //configure encoder
  try
  {
    Settings* mySettings = myEncoder->getSettingsApi();
    Settings::Name name;

    //set the profile
    name = "profile";
    mySettings->setEnum(name, PROFILE_HOME_THEATER );

    //set the bitrate, note the shorthand use of a string literal
    mySettings->setEnum("performance", PERFORMANCE_VERYSLOW );

    //want to do a single pass encode
    if( RCMODE_VBV_1PASS != mySettings->getEnum( "rcmode" ) )
    {
      mySettings->setEnum( "rcmode", RCMODE_VBV_1PASS );
    }

    //set bitrate to 500 kbps
    mySettings->setInt("bitrate", 500000 );

    //set "working_folder" note: it needs to be a valid location
    //that the encoder will be allowed to write files to
    mySettings->setStr("working_folder", "c:\\yourLocationPath");

  }
  catch(Settings::Exception& ex)
```

```
  {
    YourHandleDivXSettingsException( ex );
  }
  //done configuring encoder

  //use encoder

  //clean up when you no longer need the encoder
  EncoderInterface::destroy( myEncoder );
}
catch(DivxException& ex)
{
  //something went wrong
  YourHandleDivXExceptionFunction( ex );
}
```

3.1.4 Create an Encoder and Encode

```
try
{
  //create a pointer to the interface
  EncoderInterface* myEncoder = 0;

  //create an encoder
  myEncoder = EncoderInterface::create();

  //make sure the encoder is instantiated
  assert( myEncoder );

  //inputs to encoder
  FrameInput myFrameInput;
  FormatInfo myFormatInfo;
  FrameOutput myFrameOutput;

  //read data about your video source
  YourGetInputFormat( &myFormatInfo );//client defined

  //configure encoder to encode your source
  myEncoder->setFormatInput( &myFormatInfo )

  //set up bitstream buffer
  const int bitstreamSize=12*FormatInfo_getTotalPixels(myFormatInfo);
  myFrameOutput.bitstreamBuffer = new unsigned char[bitstreamSize];
  assert( myFrameOutput.bitstreamBuffer );
  myFrameOutput.sizeBitstreamBuffer = bitstreamSize


  int frames = YourQueryTotalNumberOfFrames();//client defined

   //see Create and Configure an Encoder section
  yourConfigureEncoder ( myEncoder->getSettingsApi() );//client defined

  for( int i=0; i<frames; i++ )
  {
      //get the next frame from the source video
      YourGetInputFrame( &myFrameInput );//client defined

      myEncoder->deliverFrame( &myFrameInput );

      while (!myEncoder->encode(&frameOutput));
```

```
        //don't write placeholder to bitstream
        if( 1 != myFrameOutput.sizeBitstream &&
            0x7F != myFrameOutput.bitstreamBuffer[0] )
        {
          yourPutBitstreamIntoContainerFunc(&frameOutput);//client defined
        }
    }
    //clean up bitstream buffer
    delete myFrameOutput.bitstreamBuffer;

    //clean up when you no longer need the encoder
    EncoderInterface::destroy( myEncoder );
  }
  catch(DivxException& ex)
  {
    //something went wrong
    YourHandleDivXExceptionFunction( ex );//client defined
  }
```

3.1.5 Create an Encoder and Perform n-pass Encode

```
  try
  {
    //create a pointer to the interface
    EncoderInterface* myEncoder = 0;

    //create an encoder
    myEncoder = EncoderInterface::create();

    //make sure the encoder is instantiated
    assert( myEncoder );

    //inputs to encoder
    FrameInput myFrameInput;
    FormatInfo myFormatInfo;
    FrameOutput myFrameOutput;

    //read data about your video source
    YourGetInputFormat( &myFormatInfo );//client defined

    //configure encoder to encode your source
    myEncoder->setFormatInput( &myFormatInfo )

    //set up bitstream buffer
    const int bitstreamSize=12*FormatInfo_getTotalPixels(myFormatInfo);
    myFrameOutput.bitstreamBuffer = new unsigned char[bitstreamSize];
    assert( myFrameOutput.bitstreamBuffer );
    myFrameOutput.sizeBitstreamBuffer = bitstreamSize


    int frames = YourQueryTotalNumberOfFrames();//client defined

    yourConfigureEncoder ( myEncoder->getSettingsApi() );//client defined

    //this configure should be in a try/catch loop, but it has been
    //omitted for clarity and brevity
    Settings* pSettings = myEncoder->getSettingsApi();
    pSettings->setEnum( "rcmode", RCMODE_VBV_MULTIPASS_1ST );
```

```
  //perform first pass
  for( int i=0; i<frames; i++ )
  {
      //get the next frame from the source video
      YourGetInputFrame( &myFrameInput );//client defined

      myEncoder->deliverFrame( &myFrameInput );

      while (!myEncoder->encode(&frameOutput));

      //this pass generates *NO* output
      //frame output will be filled with a single byte of zero

  }

  //again try/catch has been omitted
  pSettings->setEnum( "rcmode", RCMODE_VBV_MULTIPASS_NTH );

  //perform second pass
  for( int i=0; i<frames; i++ )
  {
      //get the next frame from the source video
      YourGetInputFrame( &myFrameInput );//client defined

      myEncoder->deliverFrame( &myFrameInput );

      while (!myEncoder->encode(&frameOutput));

      //don't write placeholder to bitstream
      if( 1 != myFrameOutput.sizeBitstream &&
          0x7F != myFrameOutput.bitstreamBuffer[0] )
      {
        yourPutBitstreamIntoContainerFunc(&frameOutput);//client defined
      }
  }

  //at this point you could repeat the last for loop for a 3rd pass
  //in fact you could repeat this loop as many times as you like
  //to keep refining the encode.

  //clean up bitstream buffer
  delete myFrameOutput.bitstreamBuffer;

  //clean up when you no longer need the encoder
  EncoderInterface::destroy( myEncoder );
}
catch(DivxException& ex)
{
  //something went wrong
  YourHandleDivXExceptionFunction( ex );//client defined

}
```

3.1.6 Hints for Batch Encoding

It is very easy to use the CLI to perform state saves of the encoder settings. You can use this example to perform batch encodes along with multipass encoding.

```
try
{
```

```
  //create a pointer to the interface
  EncoderInterface* myEncoder = 0;

  //create an encoder
  myEncoder = EncoderInterface::create();

  //make sure the encoder is instantiated
  assert( myEncoder );

  int batchNumber = 0;

  //first set up your batch encodes and store them in an appropriate
  //location.  It could be a file or simply a place in memory
  //depending on when you actually intend to perform the encodes.
  while( yourSetUpAnotherBatchEncode() )
  {
    std::ostrstream batchJob;

    try
    {
      //set up an encoder configuration see section labeled
      //Create and Configure an Encoder for details
      yourEncoderConfigurationFunction( myEncoder->getSettingsApi() );

      //not necessary, but useful to illustrate how to use this method
      Cli::showDisabledSettingsInNextCliRender();

      //render the cli string
      batchJob << *myEncoder->getSettingsApi();

      //store the cli string
      yourStoreBatchJob( batchJob );

    }
    catch(Settings::Exception& ex )
    {
      yourHandleSettingsException( ex );
    }
  }

  std::strstream nextJob;
  while( yourGetNextJob( nextJob ))
  {
    //shove the saved configuration into the settings
    nextJob >> *myEncoder->getSettingsApi();

    //perform the encode, see Create an Encoder and Encode
    //section for additional details
    yourReadSourceAndEncodeFunction( myEncoder );
  }

  EncoderInterface::destroy( myEncoder );
}
catch(DivxException& ex)
{
  //something went wrong
  YourHandleDivXExceptionFunction( ex );//client defined
}
```