



LAFE
TECHNOLOGIES
Supabit Compression SDK
January 20, 2006

Overview

As an industry leader in data compression, we are pleased to introduce a content-sensitive compression Software Development Kit to third parties. This SDK is composed of award winning and patented compression schemes. Unlike conventional compression approaches that apply a single algorithm to all data, the new SDK analyzes the content of each data type, and selects the optimum algorithm from a battery of compression schemes.

In addition, the component architecture of the SDK provides enormous flexibility for supporting new data types.

Another excellent feature is the ability to set quality parameters for perceptual data such as images, audio and video. This permits maximal compression with minimal perceptible loss of quality.

The SDK supports a variety of platforms including Windows, Unix, Linux and Mac OS X as well as a range of development environments such as Microsoft Visual Studio, Delphi, GCC and Eclipse.

The SDK Structure

Besides this documentation, the SDK consists of the following parts:

- A wrapper that defines the interface to the SDK
- A DLL, library and/or shared object that connects your application to the compression components.
- A folder containing compression components.
- Sample projects demonstrating the use of high and low level interfaces.

SDK distributions are targeted to specific development environments such as Microsoft Visual Studio 6 C++, Delphi 7, Linux/GCC etc.

Using the SDK

Using the SDK has been made as simple as possible. First you create a codec object. Data is passed into the object and emerges in a different form. If the input data is recognized as having been compressed by the SDK it is decompressed, otherwise the data is compressed.

Actual compression ratios depend upon the content and the desired quality level. English text can often be compressed by a factor of two or three. Multimedia compression depends upon both the original format and the allowed degree of loss, but for typical web content, ratios of five or more are not unusual. Microsoft Office documents often have compression ratios between two and ten.

The Interface

The underlying architecture is object oriented. In C and C++ the SDK is accessed via a header file and a library, while for Delphi access is via a packaged component or source code for wrapper objects.

The SDK supports two versions of the interface; high level access and low level access. Both versions of the SDK interface deals with the same two objects:

- `QCSdk` which represents the entire compression system. You need one instance of this object.
- `QCCodec` which represents a codec that is able to compress or decompress a single stream of data. You can have as many `QCCodec` objects as you need.

Delphi Access

For Delphi you create `TQCSdk` and `TQCCodec` objects.

The single `TQCSdk` object is created by calling the following function one time before using any other part of the SDK:

```
constructor Create(registryPath : string);
```

where `registryPath` is the path to the folder containing the compression components. Before using the SDK object you should check the member variable

```
fError : longint;
```

If the SDK is read to use, the value will be zero, otherwise the value will be one of the documented error return codes. An error usually means that you did not supply the path to a valid components folder.

After you have finished using the SDK, you must call the following function one time to release any memory allocated by the SDK:

```
destructor Destroy;
```

In order to create an instance of the `TQCCodec` object, you use the `CreateCodec` function of `TQCSdk`:

```
function TQCSdk.CreateCodec : TQCCodec;
```

You can use either of the following functions to see if the codec was successfully created:

```
function TQCCodec.Valid : boolean;  
function TQCCodec.Error : longint;
```

Upon failure, the function `Error` returns one of the documented error return codes.

After data compression is complete, calling the following destroys the `TQCCodec` object:

```
destructor TQCCodec.Destroy;
```

The High Level Delphi Interface

After you create an instance of a `TQCCodec` object, you must initialize it with a call to the following function:

```
function Init(quality : longint):longint;
```

where `quality` is a quality parameter where the value of 100 represents lossless compression and lower values allow for increasing degrees of compression, albeit with some loss of quality.

where `data` is the actual data and `dataLength` is the length of the data.

After all the data has been passed to the `QCCodec` object, you signal that you are finished writing data by using the following function:

```
function Finish: longint;
```

where `codec` is the `QCCodec` object used earlier. If successful, the function returns `QC_OK`. Upon failure, the function returns one of the documented error return codes.

You use the following function to extract the processed data:

```
function HRead(buffer : pchar;  
              size : longint;  
              bytesRead : longwordPtr):longint;
```

where `buffer` is a pointer to a memory data where you would like the data to be stored, `size` is the size of the storage area and `bytesRead` is the size of the data actually transferred. Call this function more than once, if necessary to extract all the data. If

successful, it returns 0 for success or 1 to signal that there is no more data to read. Upon failure, the function returns one of the documented error return codes.

The Low Level Delphi Interface

This interface is designed to give you the maximum control over compression.

The low level interface employs a `QCWrite` function that you must provide in order to transfer data. This function must have the following specification:

```
QCWrite = function(data : pchar;  
                  Length : longword;  
                  userData : pointer) : longint; cdecl;
```

where `data` points to the data to transfer and `Length` is the number of bytes of data to transfer. `userData` is a pointer to data that can be used by the `QCWrite` for any purpose you choose, for example it may be a pointer to a file descriptor. Your function must return an integer value representing the success or otherwise of the call to the function. A value of 1 denotes success. Other values indicate the nature of any error. After you create an instance of a `TQCCodec` object, you must initialize it with a call to the following function:

```
function Init(quality : longint;  
             write : QCWrite;  
             userData : pointer) : longint;
```

where `quality` is a quality parameter where the value of 100 represents lossless compression and lower values allow for increasing degrees of compression, albeit with some loss of quality; `write` is a `QCWrite` function and `userData` is a pointer to data that you wish to pass to the `QCWrite` function.

Actual data transfer to the `TQCCodec` object is performed by the following function:

```
function function Write(buffer : pchar; size : longint): longint;
```

where `data` is the data to be transferred and `length` is the length of the data.

You may call `QCCodecWrite` as many times as you wish until all the data has been transferred. Then you must call the following function to indicate that the transfer is complete:

```
function Finish: longint;
```

As the functions `Write` and `Finish` are called, they will pass output data to your `QCWrite` function to dispose of the data. The size of the data passed into your function will not be related to the size of the data written by `Write`.

C and C++ Access

In C and C++ you refer to each object by a void pointer.

The single `QCSdk` object is created by calling the following function one time before using any other part of the SDK:

```
int QCSdkCreate(const char* registryPath, QCSdk **sdk)
```

Where `registryPath` is the path to the folder containing compression components, and `sdk` is the address of the variable to assign the newly created `QCSdk` instance. If successful, `sdk` will be a valid reference to a `QCSdk` instance and the return value will be `QC_OK`. Upon failure, the function returns one of the documented error return codes, and the value of `sdk` is undefined.

After you have finished using the SDK, you must call the following function one time to release any memory allocated by the SDK:

```
void QCSdkDelete(QCSdk **sdk)
```

where `sdk` is the address of the variable containing the pointer to the `QCSdk` object. After this function returns, the variable contains a null pointer.

In order to create an instance of the `QCCodec` object, you call the following function:

```
int QCCodecCreate(QCSdk *sdk, QCCodec **codec)
```

where `sdk` is a pointer to the `QCSdk` object that you created earlier, and `codec` is the address of the variable to assign the newly create `QCCodec` instance. If successful, `codec` will be a valid reference to a `QCCodec` instance and the return value will be `QC_OK`. Upon failure, the function returns one of the documented error return codes, and the value of `codec` is undefined.

After data compression is complete, calling the following function destroys the `QCCodec` object:

```
void QCCodecDelete(QCCodec **codec)
```

where `codec` is a pointer to a variable pointing to a `QCCodec` object. This function releases any memory allocated by a `QCCodec` object. The variable `codec` is set to null upon exit.

The High Level C and C++ Interface

The high level interface works in two stages. After you have created an instance of an object, you pass the data to it with the following function:

```
QC_API int HQCCodecWrite(QCCodec *codec,  
    unsigned char *data,  
    unsigned long dataLength)
```

where `codec` is the `QCCodec` object to which you are passing the data, `data` is the actual data and `dataLength` is the length of the data.

After all the data has been passed to the `QCCodec` object, you signal that you are finished writing data by using the following function:

```
int QCCodecFinish(QCCodec *codec)
```

where `codec` is the `QCCodec` object used earlier. If successful, the function returns `QC_OK`. Upon failure, the function returns one of the documented error return codes.

You use the following function to extract the processed data:

```
QC_API int HQCCodecRead(QCCodec *codec,  
    unsigned char *data,  
    unsigned long dataLength,  
    unsigned long* returnedDataLength)
```

where `codec` is the `QCCodec` object from which you are extracting the data, `data` is a pointer to a memory data where you would like the data to be stored, `dataLength` is the size of the storage area and `returnedDataLength` is the size of the data actually transferred. Call this function more than once, if necessary to extract all the data. If successful, it returns `QC_OK` or `QC_FINISHED`. `QC_FINISHED` signals there is no more data to read. Upon failure, the function returns one of the documented error return codes.

The Low Level C and C++ Interface

This interface is designed to give you the maximum control over compression.

The low level interface employs a `QCWrite` function that you must provide in order to transfer data. This function must have the following specification:

```
int myWriteFunction(unsigned char *data, long length, *userData)
```

where `data` points to the data to transfer and `length` is the number of bytes of data to transfer. `userData` is a pointer to data that can be used by `myWriteFunction` for any purpose you choose, for example it may be a pointer to a file descriptor. Your function must return an integer value representing the success or otherwise of the call to the function. A value of 1 denotes success. Negative values indicate an error.

After you create an instance of a `QCCodec` object, you must initialize it with a call to the following function:

```
int QCCodecInit(QCCodec *codec,
               unsigned long quality,
               QCWrite *write=NULL,
               void *userData=NULL)
```

where `codec` is a pointer to the `QCCodec` object being initialized; `quality` is a quality parameter where the value of 100 represents lossless compression and lower values allow for increasing degrees of degradation; `write` points to a `QCWrite` function and `userData` is a pointer to data that you wish to pass to the `QCWrite` function.

Actual data transfer to the `QCCodec` object is performed by the following function:

```
int QCCodecWrite(QCCodec *codec, char *data, long dataLength)
```

where `codec` points to the `QCCodec` object, `data` is the data to be transferred and `dataLength` is the length of the data.

You may call `QCCodecWrite` as many times as you wish until all the data has been transferred. Then you must call the following function to indicate that the transfer is complete:

```
int QCCodecFinish(QCCodec *c)
```

where `c` is a pointer to the `QCCodec` object.

As the functions `QCCodecWrite` and `QCCodecFinish` are called, they will pass output data to your `QCWrite` function to dispose of the data. The size of the data passed into your function will not be related to the size of the data written by `QCCodecWrite`.

Return Codes

The following return codes and their meanings are employed:

0	<code>QC_OK</code> : general success
1	<code>QC_FINISHED</code> : successfully finished reading
-1	<code>QC_FAIL</code> : general failure
-2	<code>QC_EMEM</code> : failure to allocate memory
-3	<code>QC_ELDLIB</code> : failure to load library