Delphi Encryption Compendium 6.5

This is the official documentation for the Delphi Encryption Compendium 6.5 (or short DEC 6.5) library. A list of the main changes since version 5.2 can be found in the separate VersionHistory.pdf document.
Document version: 1.9 as of 1st January 2025

If you are using DEC Lite the chapters about the cipher algorithms and their demo applications are irrelevant to you. The full version of this library, including the cipher algorithms, can be found here: https://github.com/MHumm/DelphiEncryptionCompendium

Disclaimer: while we try to keep this document updated and correct, we cannot guarantee that the content is 100% error free and/or 100% complete. If you find any issues with it please tell us so we can improve it.

Contents

1 What is DEC 6.5 and what not?

DEC is a collection of cryptographic hash functions, cipher algorithms and CRC checksum routines written in Delphi and provided as open source under the Apache License 2.0. A short description of each of those algorithm categories can be found in the next chapter. It is a careful redesign of DEC 5.2 with the aim to be better maintainable, functionality wise compatible with DEC 5.2, but also to get rid of various things hindering the use on other platforms than Win32. In short, it is an internally improved version of DEC 5.2. Since the changes were so many and big and because they do influence the interface to your code they warranted a 6.0 version number. By adding XMLDOC comments to quite a lot of the methods etc. and by writing this documentation we also wanted to make this library more accessible to the casual developer. A list of the main changes can be found in the last chapter. Since DEC 6.0 bugs have been fixed and we started to add new algorithms. See separate version history document for details about the changes since then.

Our mission is to provide as many useful hash- and cipher algorithms as possible in a cross platform compatible implementation and where possible in an optimized ASM version for Win32 and Win64 as well. Just bear in mind that we are no ASM experts.

The minimum supported Delphi version is Delphi 10.1 Berlin now for all platforms.

It should be mostly Free Pascal (FPC) compatible now as somebody contributed some necessary changes, but since the main developer does not use FPC it is not tested on a regular basis. If you spot any FPC failures/issues please report them via Issues on the GitHub project.

While DEC contains sample programs and this documentation includes a little bit of cryptographic background it is not a beginner’s tutorial for properly using cryptography! The authors of this library cannot and will not take any responsibility in any way for what you do with DEC!

Additionally, DEC is not written with maximum possible speed in mind. It currently cannot use any hardware units of modern CPUs providing special commands for speeding up encryption and on platforms other than Win32 it doesn’t use assembler. While the aim should of course be to provide decent speed, the portability and maintainability of the library is at least equally important. But if volunteers want to help with coding and improving the library who knows where it can get to?

A basic set of DUnit based unit tests is being provided as well to ensure that modifications of DEC do not break anything. While not covering 100% of all possible test cases it helped us quite a lot during development as they uncovered many failures which we could fix before releasing it.

1.1 Text conventions used in this documentation

Text formatted in `Courier New and italics` references method or parameter names, properties, variables and class or unit names of DEC itself. Text formatted in *italics* but not in Courier New references Delphi RTL types or unit names.

1.2 Revision history of this document

1.0	2020/12/13	Initial revision, published with DEC 6.0
1.1	2021/01/24	Improved installation chapter Replaced chapter 5 with a list of changes to DEC 6.0 instead of 5.2 and moved this into a separate document. Replaced the progress event chapter as the technique used has been changed completely. Improved structure of the cipher algorithm chapter and explained special handling of key length for AES algorithm. Revised chapter 3.3.4 as the structure of the hash implementations changed a little and additional functionality (HMAC and PBKDF2) was added.
1.2	2021/07/03	Fixed punctuation and grammar and added extensions made for SHA3 hash and replaced hash inheritance diagram due to faults in it. Added a remark about exceptions in the unit tests.
1.3	2021/08/08	Added description of extensible output hash algorithms. Added *Known issues* chapter and updated document for V6.3.
1.4	2021/11/06	Added description of authenticated cipher block chaining modes in general and about the new *gmGCM* mode. Updated document for V6.4. Added a chapter about using TestInsight with DEC's DUnit test project.
1.5	2021/11/21	Fixed some typos of property names for GCM mode and added the new Cipher_Console_KDF demo.
1.6	2021/12/18	Improved description of CalcStream for he hashes and added description of the new overload and updated AES coverage as specific AES128, 192 and 256 classes were introduced now.
1.7	2022/10/15	Added description of BCrypt and described a compatibility issue relevant when migrating cipher code from DEC 3.x to DEC 6.x.
1.8	2024/05/08	Corrected description of TDECCipher.Init IFiller parameter.
1.9	2024/12/23	Added cipher PaddingMode, TFormat_UTF8 and warning about UTF16.

2 A short explanation of cryptography

Cryptography in general is a way of encrypting a message in such a way that only a person with the correct key can decrypt and read it. The message thus can be transferred over some insecure communication channel without enabling an unauthorized reader to read its contents.

But cryptography is more than that and DEC not only provides algorithms for encryption and decryption of text and data.

Besides some helper routines and some formatting classes DEC provides three types of algorithms which will be explained in the next subchapters.

2.1 CRC – Cyclic Redundancy Check

CRC algorithms are usually used to calculate a checksum over some data in order to be able to find out later on whether that data has been transferred correctly or stored properly on disc. Depending on the exact CRC algorithm used it can detect one or more randomly changed bits in a data stream, but the algorithm cannot correct those. Algorithms additionally being able to correct failures up to a certain degree are called error correction codes (ECC) but those are not subject of DEC.

Since it is comparatively easy to produce two messages with different contents (called a collision in the context of cryptography) but the same CRC checksum, they are not suited for cryptographic means like storing a password in a non-reversible way or guarding against malicious alternation of the data transferred. The number range of most CRC variants is simply way too small for this.

CRCs are mostly used because they can be computed quite fast. That is even more beneficial in embedded hardware where the CPU is comparatively slower than even entry level Smartphone CPUs. Many commonly used but not all CRC polynomials are initialized in such a way that calculating the CRC over the data and the appended CRC checksum leads to a result of 0. This makes checking the CRC checksum somewhat easier.

DEC contains a variety of CRC algorithms sharing the very same call interface, which makes it really easy if it should be necessary to switch the algorithm during development of an application.

2.2 Hash functions

Hash functions are a bit like CRC algorithms as far as they are mathematical one-way functions, which generate a non-reversible number from data or text given to the hash-function. The resulting number has always the same length, no matter what size the data has over which the hash has been calculated.

Since the resulting number is a quite big number, mostly 64 bit or more, the probability of collisions is significantly smaller than for CRC algorithms. Because of this hash functions are often used to prove that some text or data has not been modified or they are used to store passwords in a way which makes it impossible to recover the original clear text of the password without brute force calculation.

If hash functions are to be used for password purposes the user would enter his password, the system would calculate the hash over it and compare that to the stored hash value of that user's password. If both match the user has entered his correct password.

The brute force password breaking approach means, that one has to calculate the hash value of all permutations of allowed password characters and compare those to the stored hash value. If the hash algorithm has been properly selected and is being properly used this should be some quite time-consuming task.

Some words of caution:

1. Before using a hash function for use as one-way password storage check whether there are already known attacks or collisions for that algorithm. Do not use it when there are known collisions, as this enables to enter your system with a different password than the original one as well.
2. Do not simply hash the entered password with the algorithm and store that hash. An attacker with a precomputed table of hash values for any given input will get into your system in no time. Such tables are called rainbow tables, need quite a lot of disc space, but are readily available for most well-known hash algorithms. Now what to do? Simple: add something to the password entered and which is covered by the hash as well. Best would be a value which is different for each password record you create. You can store that value along with your hash value, as it will be needed by your password check function. Another thing to do is to calculate the hash of the hash of the hash. You get it: calculate the hash over the data several times always feeding the result of the last hash calculation as input to the new one. This also defeats the direct use of rainbow tables.
3. Pick a hash algorithm which is slow to be calculated. A brute force attack will be slowed down then, especially if combined with the methods of 2.

2.3 Cipher functions

Cipher functions are algorithms which take clear text or some binary data and encrypt it, so that somebody getting hold of that encrypted data can only make sense out of it if he has the right key to decrypt it.

There are different cipher algorithms available which have different key lengths and different cryptographic strength. Of course, they also differ in complexity and calculation time and block-based algorithms can differ in block size.

Some of them work on blocks of data with a fixed length. They are generally called *block ciphers*. For those different padding modes are available to fill up blocks when the size of the data to be encrypted is smaller than block size or not an exact multiple of it. Some of these padding modes additionally enhance security by basing the key for the next block on the encrypted output of the previous block. Other algorithms work with streams and are thus independent on block size. They are generally called *stream ciphers*.

DEC provides different padding algorithms, which can be used for all block-based cipher algorithm implementations as they are implemented in a base class. For the sake of completeness, the insecure and not recommended ECB (*E*lectronic *C*ode *B*ook) padding mode is being provided as well. DEC also provides useful wrappers which will e.g. allow working with *TStream* descendants even for block ciphers.

Before using any of the ciphers provided check whether they are suitable for your intended purpose:
1. Do you need compatibility to some other software?
2. Which security level is needed?
3. Check whether the algorithm you want to select is already known as broken! We cannot guarantee that a given algorithm is not yet broken. If we should already know about it we will document this of course.
4. If your software is to be used in different countries, check whether an algorithm of the selected strength is allowed in your target countries, as some forbid strong cryptography. I do not mean the old and luckily dead 40-bit US export cryptography limit.

2.4 Random number generator

For various cryptographic related functions good random numbers are required. Computer can only generate pseudo random numbers[1] in software (*P*seudo *R*andom *N*umber *G*enerator, PNRG). A good PNRG needs to have an even distribution of the output values.

Delphi itself includes a PNRG in the system unit, which is automatically included into all your units. This PNRG can be used by calling the `Random(x)` method. The necessary initialization by calling the `Randomize` procedure is automatically done by the Delphi RTL nowadays. If that would not be the case it would always produce the same sequence of random numbers.

DEC also contains a PNRG using a cryptographic hash function by default which makes it better suited for cryptographic purposes than Delphi's default out one.

[1] https://simple.wikipedia.org/wiki/Pseudorandomness

3 DEC explained in detail

3.1 Installation

If you fetch your copy of DEC via *Tools/GetIt* the following instructions ***do not*** apply to you.

Since DEC does not provide any components installing it is quite simple. Just unzip your downloaded DEC distribution into some empty folder. Make sure to keep the directory structure intact.

For RAD Studio/Delphi or C++ Builder:
Afterwards compile and run the *SetIDEPaths* console application from Install subdirectory. This will add the Source directory of DEC to the library paths of all RAD Studio installations found on your computer. Restart any open RAD Studio IDEs afterwards to reload their settings. The main project group is in the Source subfolder.

> DEC contains a class registration mechanism, described in chapter 3.5.
> The class registration mechanism, which is turned on by default and makes some things easier but leads to all formatting, hash and cipher classes to be compiled in even if not used. If you wish to disable this see chapter 3.4.10 DECOptions.inc about the option defines.

3.2 Known issues

The following issues/incompatibilities are currently known:

- The `Cipher_FMX` and `Hash_FMX` demos do not run on Android 32-bit release mode when optimization is turned on. Turning off optimization fixes this. Running these applications as 64-bit Android applications works in both debug and release mode.
- When C++ Builder is used, compilation should either be done using the IDE or if command line compilation is required the -DBCB parameter should be added to the command line. This is to deactivate some 32-bit asm hash implementations which are potentially incompatible with C++ Builder.
- When updating from DEC 3.x the way the key initialization of the cipher algorithms has changed. In DEC 3.x the `init` method automatically calculated a hash over the key specified and used that one as key. This will lead to incompatibilities with other cryptographic libraries and was thus removed. Now the key used is the key specified by the user when calling `init`. If you need to handle such encrypted data with the current DEC version you must calculate the hash yourself and pass that one as key. An example of how to do this might look like this:

```
uses
  DECCipherBase, DECCiphers, DECHash;
var
  Cipher     : TCipher_Blowfish;
  InitVector : TBytes;
  Hash       : THash_RipeMD256;
  HashResult : TBytes;
begin
```

```
Cipher := TCipher_Blowfish.Create;
try
  Hash := THash_RipeMD256.Create;
  try
    HashResult := Hash.CalcBytes('My key');
  finally
    Hash.Free;
  end;

  // Sample init vector only, as this should be unique
  SetLength(InitVector, 8);
  FillChar(InitVector[0], Length(InitVector), #65);

  Cipher.Init(HashResult, InitVector);

  // Put cryptographic operation like encode or decode here
finally
  Cipher.Free;
end;
```

3.3 General structure

DEC 6.5 contains the following parts/directories:

\Docs
Contains all the documentation, including the one you are currently reading. If you need help using DEC please look at the provided docs first.

\Source
This directory contains the units of DEC in source code form, so everything is transparent to you.

File	Purpose/Contents
`DECBaseClass.pas`	Contains the root class of all DEC classes and the class registration mechanism, which is explained in chapter 0 The class registration mechanism.
`DECTypes.pas`	This one contains just a few type declarations.
`DECCipherBase.pas`	Contains the root class all cipher classes inherit from, providing the basic infrastructure used by the individual cipher classes.
`DECCipherModes.pas`	This unit contains the class implementing the block chaining modes like CBC. Normally only used internally.
`DECCipherModesGCM.pas`	Implements the GCM specific block chaining mode. It is not used directly. Usage is via `DECCipherModes.pas`.
`DECCipherFormats.pas`	The class contained in this unit provides various convenient ways to feed the data which shall be encrypted or decrypted to DEC. For instance, it provides methods to feed the data as *string* or as *TStream*.
`DECCipherPaddings.pas`	Contains the abstract base implementation for all cipher padding implementation classes and the available derived padding implementation classes.
`DECCipher.pas`	This unit contains all the different cipher algorithm implementations.
`DECCipherInterfaces.pas`	Contains interfaces for the methods provided by `TDECCipherModes` and the additional methods and properties used by the authenticated encryption algorithms such as GCM.
`DECDataCipher.pas`	This unit contains various precalculated/ constant initialization- or permutation tables for the different cipher algorithms.
`DECData.pas`	This unit contains precalculated/ constant initialization- or permutation tables used in both cipher and hash algorithms.
`DECDataHash.pas`	This unit contains various precalculated/ constant initialization- or permutation tables for the different hash algorithms.
`DECHashBase.pas`	Contains the root class all hash-algorithms inherit from, providing the basic infrastructure used by the individual hash classes.
`DECHash.pas`	This unit contains all the different hash algorithm implementations.
`DECFormatBase.pas`	Contains the root class all format-algorithms inherit from, providing the basic infrastructure used by the individual format classes.
`DECFormat.pas`	This unit contains all the different format algorithm implementations.
`DECCRC.pas`	This unit contains various CRC implementations.
`DECUtil.pas`	This unit contains most if not all the exception declarations used in DEC and various utility methods to swap bytes in bigger datatypes, to protect memory after use and a convenient little method to convert the contents of a *TBytes* array to *RawByteString*, usually used for debugging purposes.
`DECRandom.pas`	This unit contains the cryptographic pseudo random number

	generator.
DECHash.asm86.inc	This include file contains x86 assembler implementations of most of the hash-algorithms. These are being used when the *NO_ASM* define in *DECOptions.inc* is turned off and the target is Win32.
DECHash.SHA3_mmx.inc	This include file contains the MMX optimized 32 Bit assembler implementation of the SHA3 permutation. This is being used when the *NO_ASM* define in *DECOptions.inc* is turned off and the target is Win32.
DECHash.SHA3_x64.inc	This include file contains the optimized 64 Bit assembler implementation of the SHA3 permutation. This is being used when the *NO_ASM* define in *DECOptions.inc* is turned off and the target is Win64.
DECOptions.inc	Include file with various compiler defines controlling how DEC works in certain cases. For details see chapter 3.4.10 DECOptions.inc.

\UnitTests

In order to ensure that DEC properly works and that any change somebody should make to its source code still produces a properly working version of DEC we created a bunch of DUnit unit tests. Additionally, we try to be DUnitX compatible with our tests. We currently simply prefer DUnit because DUnit is included with older Delphi versions already and it has a nice and helpful GUI runner. We did not yet manage to get the GUI runner of DUnit X to work. DUnit also hast a test case skeleton generator built into an IDE wizard. If you want to use TestInsight with this you find instructions in chapter 3.6 Unit Test TestInsight integration.

You should be able to load the *DECDUnitTestSuite* or the *DECDUnitXTestSuite* Project, compile and run it. You can select between *DUnit* and *DUnitX* by enabling or disabling the *DUnitX* define. It is located in *defines.inc* in the unit test projects. In order to enable it remove the . in front of the $ sign. To disable it, add the . again.

With this unit test project, you should be able to verify that the version of DEC you are using passes all tests. The tests mostly cover the basics only so these are not a 100% guarantee that DEC is bug free, but those tests already helped us quite a lot while reshaping DEC!

> Caution: some of the unit tests do test for exceptions. When they are being run from the IDE the usual message dialogs about occurring exceptions are shown and the unit test is halted until the dialog is answered. This is expected! The option to ignore that specific exception can be used to suppress these dialogs.

Those users knowing the old distribution might know the old test application using the test vectors (test data) from a text file. We not only converted this hard to read application into unit tests, we also added tests for areas not covered yet, e.g. for the CRC routines.

\Demos

This directory contains some simple demo projects aimed to help you getting started with DEC. A list and descriptions of the provided demos can be found in chapter 4 Demos.

3.4 Using DEC

3.4.1 The DEC base class

All classes of DEC derive from a common base class *TDECObject*. This class is implemented in *DECBaseClass.pas*. Most of its methods are class methods, so they can be directly called on a class reference without requiring an object reference. But of course, they can be called on a proper object reference as well. Most deal with DECs class registration mechanism, which is described in detail in chapter 3.5 The class registration mechanism. You usually do not have much if any contact with this class itself unless you work on the DEC code base.

Method	Purpose
Identity	This class method delivers a number which should be unique of a class derived from this base class. You can store this number in a file to encode the hash- or cipher algorithm used for creating this file and by using the appropriate registration mechanism you can later on quite easily create the required hash or cipher instance needed based on this identity.
FreeInstance	This method is only available if use ASM routines in DECOptions.inc has been turned on. It has to do with safely clearing memory on its release by overwriting it with zeroes.
SelfTest	While knowing what a self-test generally is it's not clear what exactly was meant with this. It might get removed in a subsequent version.
RegisterClass	Adds the class reference to the global list of registered classes which is passed as parameter. This method is usually not called in user code, as each relevant DEC class is already being registered in the initialization section of the unit implementing the class.
UnregisterClass	Removes the class reference from the global list of registered classes which is passed as parameter. This method is usually not called in user code, as each relevant DEC class is already being unregistered in the finalization section of the unit implementing the class.
GetShortClassNameFromName	Returns the short class name of a class name being passed as parameter. For instance, the short class name of *TCipher_Skipjack* is *Skipjack*.
GetShortClassName	Returns the short class name of this class.

3.4.2 Using the formatting routines

Why do we start our tour through the DEC libraries with the formatting routines? That's simple: because they can be used together with all other categories of routines. They are being used to format data in various ways and to pass that to the other methods and functions or to convert the data returned by those into one of the provided standard formats. And sometimes it's simply helpful to have a quick way to display a hexadecimal representation of returned binary data to check something while debugging.

All the provided formatting classes have a common ancestor: `TDECFormat` which is implemented in the *DECFormatBase.pas* Unit and all of those provide all the public methods of `TDECObject` as well as described in the preceding chapter.
The formatting classes provide their complete functionality in form of class procedures and class functions, so you never need to create an instance of a formatting class. The implementations are in *DECFormat.pas.*

The following methods are being provided:

Method	Purpose
Encode	Formats a given byte array into the format of the formatting class. The output is a byte array. Two deprecated overloads for use with RawByteString and untyped data are being provided as well. These overloads have a RawByteString as result.
Decode	Formats a byte array given in the format of the formatting class back into the original format. Output is a byte array.
IsValid	Checks whether the data passed to it is valid for that particular formatting. This is useful as some formats only allow a certain range of input values.
UpCaseBinary	This method works similar to the *UpCase* routine of *system.pas* with the following differences: it only works for the character range a-z and input and output are not a char each but a byte instead.
TableFindBinary	This method looks for the first occurrence of a given byte within a given byte array. If the byte has been found the index within the byte array is being returned, otherwise -1 is returned.
FilterChars	Delivers all chars which are allowed to be passed to the conversion methods when somebody wants to convert into this format. Result is empty if there is no restriction on allowed chars.

List of provided formatting classes

Format class	Format / purpose
TFormat_Copy	This class doesn't apply any formatting change on the data passed in. It can be used in places where a formatting class is being expected but when you do not want to have any format change applied.
TFormat_HEX	Converts the input into an upper-case hexadecimal representation. One byte of the input will be converted into a two bytes hex representation. Be aware that Unicode strings are UTF16 encoded, which means that each character you see in the string consists at least of 2 bytes, even if it is in the ASCII range. The 2nd byte will simply be 0 in that ASCII case. The letters A-F in the hexadecimal representation will be uppercase A-F characters.
TFormat_HEXL	The same as format *TFormat_HEX*, just with lower case letters a-f.
TFormat_Base16	Alias for *TFormat_HEX* for compatibility reasons.
TFormat_Base16L	Alias for *TFormat_HEXL* for compatibility reasons.
TFormat_DECMIME32	This is a special format created by Hagen Reddmann, the original author of DEC. We do not recommend using this one, as it will only be compatible with DEC itself!

`TFormat_Base32`	This format converts data into char sequences only consisting A-Z and 2-7. They are filled up with = to multiples of 8 bytes. The specification for this can be found in RFC4648[2].
`TFormat_Base64`	This format converts 8-bit bytes into some code page invariant ASCII representation. Means: each input byte will be encoded in such a way that it can be written with an ASCII character which is encoded the same on all ASCII DOS or ANSI codepages since it belongs to the 7-bit ASCII range. While this means you can transmit such binary data with an ordinary e-mail application within the message body it also means, that data encoded with this scheme requires a bit more space as from each byte of the Base64 representation only the lower 7 bits can be used.
`TFormat_MIME64`	Alias for `TFormat_Base64` for compatibility reasons.
`TFormat_Radix64`	This is a variant of `TFormat_Base64` used in the OpenPGP context. It is basically a `TFormat_Base64` with an added 24-bit checksum.
`TFormat_PGP`	Alias for `TFormat_Radix64` provided for compatibility reasons but deprecated.
`TFormat_UU`	The UUEncode formatting is slightly similar to Base64. From the name it is Unix to Unix and is being used to transfer binary data via e-mail. 24 bits of input are being re-encoded into 4x 6 bit. For this only the ASCII characters 33 to 96 are being used.
`TFormat_XX`	This format is quite similar to `TFormat_UU`. It just further reduces the characters used to encode the binary data to just the letters, digits and the plus and minus sign. This shall reduce the danger that some application somehow interprets special characters as something else and thus ruins the encoding.
`TFormat_ESCAPE`	This is a variant of the Hex format but with the addition that certain characters are treated as escape characters. These are especially the escape sequences found in C-style languages used to denote line breaks or carriage returns etc. This is an incomplete list of the escape characters: \a \b \t \n \v \f \r
`TFormat_BigEndian16`	Swaps the byte order of the data passed in. Example: 1 2 3 4 becomes 2 1 4 3. If the data contains an odd number swapping will still be performed and no exception will be raised but the data will be declared invalid when using `IsValid` to check whether it is valid for this algorithm. Use should be avoided for scripts using mainly 8-bit character codes due to lowered security in this case. If possible convert to UTF8 and use that one.
`TFormat_BigEndian32`	Swaps the byte order of the data passed in. Example: 1 2 3 4 becomes 4 3 2 1. If the length of the data cannot be divided by 4 without reminder swapping will still be performed and no exception will be raised but the data will be declared invalid when using `IsValid` to check whether it is valid for this algorithm. Use should be avoided for scripts using mainly 8- or 16-bit character codes due to lowered security in this case. If possible convert to UTF8 and use that one.
`TFormat_UTF8`	Converts the data passed in into an UTF8 encoded string.
`TFormat_UTF16`	This is an alias of `TFormat_BigEndian16`, but with its own identity value. Use should be avoided for scripts using mainly 8-bit character codes due to lowered security in this case. If possible convert to UTF8 and use that one.

[2] https://datatracker.ietf.org/doc/html/rfc4648

In addition to the methods listed above, the formatting classes do have this class variable, which they inherit from their base class:

`ClassList` – this public class variable contains the hash algorithm registration list, which provides access to all hash classes. For details about the registration mechanism see chapter 3.5 The class registration mechanism.

3.4.3 Using the CRC algorithms

The CRC algorithms are located in the *DECCRC.pas* Unit. There are two sorts of routines being provided. The first and easier to use ones calculate the CRC value in one single step and are thus most suited for smaller amounts of data to be processed, as implementing a progress display during their runtime is not possible.

There exist the following 4 variants:

- `CalcCRC` with a buffer as parameter. Pass in any array or TBytes type you like and pass a parameter telling how many bytes from that buffer, starting at its beginning, go into the CRC calculation.
- `CalcCRC` with a callback as parameter. As callback you need to pass a method having an untyped buffer as var parameter and an Int64 typed size parameter specifying how many bytes from the beginning of your buffer parameter will go into the CRC calculation. The `CalcCRC` routine will call your callback as often as needed until it has `Size` bytes for calculating the CRC.
- `CRC16` is a variant which does not let you specify which algorithm to use. It will use the IBM/ARC/MODBUS RTU CRC16 algorithm.
- `CRC32` is a variant which does not let you specify which algorithm to use. It will use the CRC32-CCITT algorithm. It works on an untyped `Buffer` parameter and processes `Size` bytes of that buffer, beginning at the start of it.

The other sorts of routines split the CRC processing into several steps and thus they give you finer control about what to do at a given place in your code.

> ⚠ Caution: when using the CRC16 or CRC32 functions in a multithreaded application, you need to call `CRCInitThreadSafe` first!

3.4.4 Using the hash algorithms

3.4.4.1 Base structure of the hash algorithms

The hash algorithm classes have a mostly common API. Parts of this API are implemented in abstract ancestor classes. The following diagram illustrates this, where abstract classes are written in *cursive*. Any spaces within class names are in fact _ chars, which are not properly rendered because of a lower than usual line height used to make the tree lines connect to each other.

```
TDECObject
├─TDECFormat
│ ├─TFormat_HEX
│ │ ├─TFormat_HEXL
│ │ ├─TFormat_DECMIME32
│ │ └─TFormat_Base64
│ │   └─TFormat_Radix64
│ ├─TFormat_UU
│ │ └─TFormat_XX
│ ├─TFormat_ESCAPE
│ ├─TFormat_BigEndian16
│ ├─TFormat_BigEndian32
│ ├─TFormat_BigEndian64
│ └─TFormat_Copy
└─TDECHash
  └─TDECHashAuthentication
    ├─TDECHashExtended
    │ ├─TDECHashBit
    │ │ └─THash_SHA3Base
    │ │   ├─THash_SHA3_224
    │ │   │ └─THash_Keccack_224
    │ │   ├─THash_SHA3_256
    │ │   │ └─THash_Keccack_256
    │ │   ├─THash_SHA3_384
    │ │   │ └─THash_Keccack_384
    │ │   ├─THash_SHA3_512
    │ │   │ └─THash_Keccack_512
    │ │   └─THash_ShakeBase
    │ │     ├─THash_Shake128
    │ │     └─THash_Shake256
    │ ├─THash_MD2
    │ ├─THashBaseMD4
    │ │ ├─THash_MD4
    │ │ ├─THash_MD5
    │ │ ├─THash_RipeMD128
    │ │ ├─THash_RipeMD160
    │ │ ├─THash_RipeMD256
    │ │ ├─THash_RipeMD320
    │ │ ├─THash_SHA0
    │ │ │ ├─THash_SHA1
    │ │ │ └─THash_SHA256
    │ │ │   └─THash_SHA224
    │ │ └─THash_Tiger
    │ ├─THash_SHA384
    │ │ └─THash_SHA512
    │ ├─THashBaseHaval
    │ │ ├─THash_Haval128
    │ │ ├─THash_Haval160
    │ │ ├─THash_Haval192
    │ │ ├─THash_Haval224
    │ │ └─THash_Haval256
    │ ├─THash_Panama
    │ ├─THashBaseWhirlpool
    │ │ ├─THash_Whirlpool0
    │ │ ├─THash_WhirlpoolT
    │ │ └─THash_Whirlpool1_
    │ │   └─THash_Whirlpool1
    │ ├─THash_Square
    │ ├─THashBaseSnefru
    │ │ ├─THash_Snefru128
    │ │ └─THash_Snefru256
    │ └─THash_Sapphire
    └─TDECPasswordHash
      └─THash_BCrypt
```

The base classes *TDECHash* is implemented in the *DECHashBase.pas* unit. The *TDECHashAutentication*, *TDECHashExtended* and *TDECPasswordHash* classes are implemented in the *DECHashAuthentication.pas* unit.

The *TDECHashAuthentication* class contains all key deviation, mask generation function, hash message authentication and password based key deviation function implementations. They can only be used from a class which actually implements a hash algorithm, this means it must be a class descending from *TDECHashAuthentication* in some way. This class does not implement an interface itself because all methods it provides are class methods. This means that these methods do not need to be called on an object instance. They can be called on the class itself.

All hash algorithms specifically suited for password hashes should inherit from the *TDECPasswordHash* class, which implements the *IDECHashPassword* interface. Password hash algorithms are designed to be slow to calculate in order to make breaking passwords by brute force expensive.

In order to make it easy to find out whether a given hash class is specifically designed for password hashing, all hash classes contain a class function named `IsPasswordHash`. This method checks, whether the class inherits from `TDECPasswordHash`.

Most hash algorithms work on messages with whole 8-bit bytes as contents. Some hash classes, most notably the SHA3 family and derived algorithms, can process messages ending on fractions of bytes or even a few bits alone. Those algorithms inherit either from the `TDECHashBit` class directly or if they are from the SHA3 family, from `TDECSHA3`. The `TDECHashBit` class is implemented in the `DECHashBitBase.pas` unit and it provides the `IDECHashBitsized` interface, which is an extension of the `IDECHash` interface. The `TDECHashBit` class only adds one property: `FinalByteLength`. If this is not 0 (its default value), it denotes the number of bits of the final byte of the message which shall be included in the hash calculation. If the method used for hashing contains a size parameter, this size needs to include such a partial last byte and the `FinalByteLength` specifies that it is a partially used byte. For SHA3 based algorithms the padding to fill up this last byte is automatically included in the algorithm so one cannot specify a padding value for these algorithms.

All operations not useful for password hash classes have been moved into the `TDECHashExtended` class. This way `TDECPasswordHash` can inherit directly from `TDECHashExtended`'s base class `TDECHashAuthentication` and will not provide the methods which operate on files and streams. The methods moved to the `TDECHashExtended` class are the ones calculating a hash from a stream or over a file, both usually containing more data than the password hash algorithms are designed for. An interface named `IDECHashExtended` is available for the `TDECHashExtended` class as well.

In addition to the basics required for password hash implementations, the `TDECPasswordHash` class provides the basis for Crypt/BSD style output as well, where password hashes are stored in a text file together with a unique algorithm identifier, the hashed salt and other parameters relevant for the specific algorithm. Some password hash algorithms will provide the necessary implementations, but only if they have such a unique Crypt/BSD ID. Those classes providing such an ID can be retrieved via the `ClassByCryptIdentity` method, which searches for the registered class type implementing the unique ID passed as parameter. This requires that the class registration mechanism for hashes is turned on. For details about this mechanism see chapter *3.5 The class registration mechanism*.

All hash classes provide all the public methods of `TDECObject` as well as described in chapter *3.4.1 The DEC base class.*

3.4.4.2 Methods for using the hash classes

Since all the hash classes inherit from *TDECHashBase*, they mostly share a common API for using them. Exceptions to this rule will be explained in the next chapter.

Method	Purpose
`BlockSize`	Returns the size of a data block in bytes. The data given to the hash algorithm is being processed in blocks of this size internally and if the data does not fill the last block completely it will be automatically filled with the `PaddingByte` specified.
`Calc`	Calculates the hash value over a chunk of data.
`CalcBuffer`	Calculates the hash value over a given buffer of data. The size of the buffer in bytes needs to be specified as well and the result is the calculated hash value as *TBytes* array.
`CalcBytes`	Calculates the hash value over a given *TBytes* buffer of data. The result is the calculated hash value as *TBytes* array.
`CalcString`	Calculates the hash value over a string. There exist two overloads: one for Unicode strings and one for *RawByteStrings*. Both have an optional parameter where you can pass a formatting class. The formatting will be applied to the calculated hash value, e.g. you can get the hash value hex formatted this way for instance. If no formatting is being passed, the returned string is simply the interpretation of the calculated hash value bytes as a string. In case of an UnicodeString, which is being returned here, the result might be undesired.
`ClassByIdentity`	If one knows the numeric identify value of a given hash-implementation class, this method can be used to retrieve the class reference from the registration list. So, this method is useful if one wants to create an object-instance of a certain hash-implementation class for which one knows the identity value (e.g. because such a value is stored in a file header). If the queried class cannot be found in the registration list an `EDECClassNotRegisteredException` exception will be raised.
`ClassByName`	Searches for a class with the name given as parameter in the class registration list. If a matching class is found, the class reference is returned. This can be used to create an object of that class. So, this method is useful when one wants to create an object of a certain hash-algorithm implementation but only knows the name of the hash class as string. If the queried class cannot be found in the registration list an `EDECClassNotRegisteredException` exception will be raised.
`DigestAsBytes`	Returns the calculated hash value as TBytes byte array.
`DigestAsRawByteString`	Returns the calculated hash value as a *RawByteString*. If one of the formatting classes is being passed via the optional `Format` parameter this formatting is being applied to the return value, e.g. you can get the hash value hex formatted this way for instance. If no formatting is being passed, the returned string is simply the interpretation of the calculated hash value bytes as a string.

DigestAsString	Returns the calculated hash value as a Unicode string. If one of the formatting classes is being passed via the optional *Format* parameter this formatting is being applied to the return value, e.g. you can get the hash value hex formatted this way for instance. If no formatting is being passed, the returned string is simply the interpretation of the calculated hash value bytes as a string. In case of an UnicodeString, which is being returned here, the result might be undesired.
DigestSize	Returns the length of a calculated hash value in bytes.
Done	Finalizes hash calculation and clears the buffers used in a safe way to prevent stealing of data. Must be called at the end of each hash value calculation.
Init	This method needs to be called directly before each hash value calculation. It initializes the properties of the algorithm and clears all required buffers with default values.
IsPasswordHash	This class method returns true if the class on which it is called is a class specifically designed for generating password hashes. The implementation here always returns false.

In addition to the methods listed above the hash classes do have this class variable, which they inherit from their base class:

ClassList this public class variable contains the hash algorithm registration list, which provides access to all hash classes. For details about the registration mechanism see chapter 3.5 The class registration mechanism.

All classes also have this common property:

PaddingByte the value assigned to this byte is being used to fill up data passed to the hash algorithm if the data does not completely fill the last block. Means: if the size of the data passed cannot be divided by *BlockSize* without reminder. In case of algorithms having specific requirements about the last byte (like SHA3) this is of course ignored.

The `TDECHashAuthentication` class implements the following class methods:

Method	Purpose
`HMAC`	Creates a message authentication code over a given text. It is based on rfc2202. Parameters are the secret `Key` both parties shared securely at some point and the `Text` the authentication code shall be calculated on. Returned is the calculated authentication code.
`IsPasswordHash`	Returns true if this class implements a hash algorithm particularly designed for hashing passwords, means: if the class inherits from `TDECPasswordHash`. Since `TDECHashAuthentication` is not to be used directly "this" means the descending class implementing the actual hash algorithm.
`KDF1, KDF2, KDF3`	All these key deviation methods exist as the same overloads with the same parameters. One overload each takes untyped parameters and the other one *TBytes* based parameters. More details are given in chapter 3.4.5.2 KDF1, KDF2, KDF3.
`KDFx`	Key deviation method similar to KDF1-KDF3, but not based on any official standard. Developed by the original author of DEC.
`MGF1`	MGF1 is a mask generation function defined in the Public Key Cryptography Standard #1 published by RSA Laboratories[3][4] More details are given in chapter 3.4.5.1 MGF1.
`MGFx`	Key deviation method similar to MGF1, but not based on any official standard. Developed by the original author of DEC.
`PBKDF2`	This is a key deviation function based on a user specified password, a salt value and an iteration count. Returned is the generated password hash value. It is based on RFC 2898 and PKCS #5 and uses the same algorithm as HMAC.

The `TDECHashExtended` class adds the methods listed in the table below. They got moved there from the `TDECHashBase` class. They are not in the base class anymore because the `TDECHashPassword` class should not contain them as they are not useful for password hashing.

Method	Purpose
`CalcFile`	Both overloads of this method calculate the hash value over the contents of a file. The file is specified by its path and file name. One of the overloads returns the hash value as a *RawByteString* return value and for this it contains an optional format parameter for passing a formatting class used to format the output. The other one contains a *TBytes* parameter where it will return the calculated hash value in. There cannot exist overloaded methods in Delphi which only differ in the data type of the return value. The last parameter is optional. You can supply a callback method here which will be called by the method to report calculation progress. This is especially useful for big sized data, as you can display the progress of the operation via this callback method. Be aware though, that if the hash method is running in the application main thread any message pump required for updating display controls might not be run. So, if calculating hash values over large

[3] https://en.wikipedia.org/wiki/Mask_generation_function#MGF1
[4] MGF1 is defined in the IEEE P1363a and PKCS#1 v2.1 standards

	amounts of data and wishing to display progress you should run the hash calculation in a separate thread. This allows display updates to work and keeps your main thread responsible. Both call `Init` and `Done` internally.
CalcStream	For this; three overloads exist: Two overloads of this method calculate the hash value over the contents of a stream. The stream may be a file stream, a memory stream or any other kind of stream. You have to specify the size of the stream as a parameter. These two overloads call Init and Done of the hash class each, so they can only be used if all data, over which the hash shall be calculated, is already contained in the stream when it is being called. One of the overloads returns the hash value as a *RawByteString* return value and for this it contains an optional format parameter for passing a formatting class used to format the output. The other one contains a *TBytes* parameter, where it will return the calculated hash value in. There cannot exist overloaded methods in Delphi which only differ in the data type of the return value so it had to be implemented as var-parameter each. The last parameter of these two is optional. You can supply a callback method here which will be called by the method to report calculation progress. This is especially useful for big sized data, as you can display the progress of the operation via this callback method. Be aware though, that if the hash method is running in the application main thread any message pump required for updating display controls might not be run. So, if calculating hash values over large amounts of data and wishing to display progress you should run the hash calculation in a separate thread. This allows display updates to work and keeps your main thread responsible. The third overload does not return a calculated hash value and thus it does not call Init or Done automatically. Init always must be called once for the hash instance before calling this method for the first time or after Done has been called on the instance (means: if you reuse the instance). Done is called when the last parameter *DoFinalize* is *true*. The calculated hash value can be obtained by calling *DigestAsBytes*, *DigestAsString* or *DigestAsRawByteString*, after calling CalcStream with true for the last parameter. It possesses the same optional callback parameter for implementing progress display. If this possibility shall not be used just pass nil for this parameter.

The *TDECHashBit* class implements the following property, which is also accessible via the *IDECHashBitsized* interface:

Property	Purpose
FinalByteLength	Defines the number of bits of the final byte of the message which will be considered when calculating the hash of a message with an algorithm which can define the processed message length in bits. Most notable algorithm for this is SHA3. A value of 0 means that all bits of the last byte are included in the message calculation. SHA3 fills automatically up the last byte as necessary (padding) when the final byte length is lower than 8 bits. The padding cannot be influenced by the caller as this is standardised with the algorithm.

The following hash algorithms inherit from this: all *SHA3* variants, *Shake128* and *Shake256*.

The *TDECHashPassword* class is the base class for all specialized password hashing classes. It implements the following property, which is also accessible via the *IDECHashPassword* interface:

Property	Purpose
Salt	Most, if not all, password hashing algorithms have a salt value, which is being applied to the password during hashing. The salt should be different for each stored password, as this prevents some attacks on the password scheme used or at least makes them harder. By default the value of this salt property is securely overwritten after each hash calculation! This includes calls to *GetDigestInCryptFormat* and *IsValidPassword*!

And these class functions:

Function	Purpose
MaxPasswordLength	Some password hash algorithms, like BCrypt, only allow a limited number of bytes. This class function returns the maximum number of bytes supported by this password hash algorithm.
MaxSaltLength	Specifies the maximum length in byte allowed for the salt.
MinSaltLength	Specifies the minimum length in byte allowed for the salt.
ClassByCryptIdentity	Class method for retrieving a class type for a hash algorithm, specified by its Crypt/BSD unique identifier. For example, passing *'2a'* as parameter would return *THash_BCrypt*, if that one is registered in the class registration mechanism. If no matching class type for a given ID can be found a *EDECClassNotRegisteredException* exception will be thrown.
GetDigestInCryptFormat	Calculates the Crypt/BSD style data record for a password, which would be stored in a text file when creating a new password for a user. The *SaltIsRaw* Boolean parameter specifies, whether the salt passed is in Crypt/BSD format (= false) or if it is in raw bytes value. As last parameter the name of the formatting class used for the Crypt/BSD style data parameter needs to be supplied. It was designed this way to not require the *TDECFormat* unit in the uses-clause. Different password hash algorithms use different

	formats for their Crypt/BSD style data storage anyway. For the method two overloads exist: one with string type for the password, the other one with a TBytes typed password parameter.
`IsValidPassword`	This method returns true, if the password provided as parameter is the correct password for the Crypt/BSD style formatted data passed as well. As 3rd parameter the name of the formatting class used for the Crypt/BSD style data parameter needs to be supplied. It was designed this way to not require the *TDECFormat* unit in the uses-clause. Different password hash algorithms use different formats for their Crypt/BSD style data storage anyway. For the method two overloads exist: one with string type for the password, the other one with a TBytes typed password parameter.

The *THash_ShakeBase* class implements the following property, which is also accessible via the *IDECHashExtensibleOutput* interface:

Property	Purpose
HashSize	Defines the length of the generates hash value in byte. A length of zero is not allowed!

The following hash algorithms inherit from this: *THash_Shake128* and *THash_Shake256*.

3.4.4.3 Exceptions to the common API for hash classes

There are a few hash classes which provide additional API methods or properties. The following paragraphs list those. Be aware that those additional methods or properties are not accessible via the *IDECHash* interface.

THash_Snefru128, THash_Snefru256, THash_Haval128, THash_Haval160, THash_Haval224, THash_Haval256, THash_Tiger and its alias THash_Tiger192:
All of them have an additional property *Rounds*. Those algorithms use several rounds of calculation where the result of the preceding round will be the input for the next round. This property sets the number of rounds to use.

These algorithms also have the class methods *GetMinRounds* and *GetMaxRounds* to be able to determine the minimum and maximum allowed values for the *Rounds* property.

Algorithm	Min rounds	Max rounds	Behaviour when exceeding value specified
THash_Tiger	3	32	*Rounds* is either set to the minimum value of 3 or the maximum value of 32.
Haval (all variants)	3	5	*Rounds* depends on the *DigestSize* set. For a *DigestSize* of 20 or lower it will be 3, for *DigestSize* 28 or lower but bigger than 20 it will be 4 and for values bigger 28 it will be 5.
Snefru (all variants)	2	8	*Rounds* is set to 8.

THash_Sapphire
This one has a property *RequestedDigestSize*. With this you can define how many bytes of the calculated hash value will be returned via the *DigestAsBytes* method. The *Digest* method is not affected by this. Values bigger 64 do not make sense, as the hash value is only 64 byte long. If the *RequestedDigestSize* is set to 0 the default value of 64 byte is being used.

THash_Bcrypt
This one has a property *Cost*. It is quite similar like the *Rounds* property of those hash classes containing a *Rounds* property. The difference lies in the encoding. While *Rounds* is the direct number of calculation-cycles the algorithm performs on the data to be hashed, *Cost* specifies rounds as 2^{Cost}.

3.4.4.4 Differences between SHA3 and Keccack

Some people think that SHA3 and Keccack refer to the exact same thing. But that is not completely true. While Keccack is the algorithm which later became the SHA3 standard, there is one little difference in the way how the padding is handled. This leads to different output obviously and thus if one needs to interact with other software one needs to carefully check if that software really implements SHA3 or Keccack. Some other libraries do not seem to have noticed that difference or were only updated later on, so there still might be some implementation sticking around which is being called SHA3 but is Keccack instead.

3.4.4.5 Interfaces for the hash classes

If you like to use the good programming habit of programming against interfaces instead of using concrete classes you can do so. There are several interfaces for different aspects/kinds of hashes provided.

Interface	Implemented by	Purpose
IDECHash	*TDECHashBase*	Contains all public methods and properties of *TDECHashBase*. The exceptions are the class methods, as interfaces in Delphi do not support those. This interface can be used for programming against interfaces with all hash algorithms except for the extensible output length ones where this is not sufficient.
IDECHashBitsized	*TDECHashBit*	Contains the property used to specify how many bits of the last byte shall be included in the hash calculation. The interface inherits from the *IDECHash* interface.
IDECHashExtensibleOutput	*THash_ShakeBase*	Contains the property used to specify the size of the generated hash in byte, which needs to be specified for the extensible output length hash algorithms. The interface inherits from the *IDECHash* interface.
IDECHashRounds	*THash_Tiger* *THashBaseHaval* *THashBaseSnefru*	Contains the *Rounds* property which specifies the number of rounds used for the calculation and the *MinRounds* and *MaxRounds* methods for determining the minimum and maximum allowed values.

3.4.5 Using the key deviation algorithms

Key deviation algorithms[5] are used for deriving further keys from already existing keys without being able to determine the key from which the derived one was derived of. A simple scheme for deriving a 2nd key from of a first one could be to calculate the hash sum of the first key via some well-defined hash algorithm. If a 3rd key is needed, one would simply calculate the hash sum on the 2nd key, using the same algorithm. That way nobody can tell whether different keys descend from each other by just looking at the keys.

[5] https://en.wikipedia.org/wiki/Key_derivation_function

One use case for this class of algorithms is to generate a password hash value using a hash algorithm not originally developed for password hashing. Without applying the key deviation function the original hash algorithm would create a too week hash value for safe use as a password hash.

Another use case is to derive keys for additional purposes from a password so these keys are tied to the user login password.

Another property of those key deviation algorithms is, that one can specify the size of the key resulting from the calculation.

All key deviation methods provided by DEC are class methods of the *TDECHash* class.

3.4.5.1 MGF1

This key deviation method has been specified in RFC 2437 as PKCS #1[6]. It is a variant of the *KDF1* algorithm defined in the ISO 18033-2:2004 standard. DEC provides two overloaded class methods for this. The first one takes an unspecified data parameter followed by a 2nd parameter specifying the length of the data given in the first one in byte. The second one takes a *TBytes* array for the input data.

Both methods have a parameter *MaskSize*. It specifies the length of the generated output in bytes. The output is a *TBytes* array for both variants.

3.4.5.2 KDF1, KDF2, KDF3

These three algorithms are relatives. The difference between *KDF1* and *KDF2* is whether the calculation loop counter runs from 0 to round – 1 or from 1 to rounds. *KDF3* is like *KDF1* but two calculation steps are reversed.

For each of these algorithms two overloads are being provided. The first variant has an untyped *Data* parameter for specifying the key from which a new one shall be deviated. Because this parameter is untyped a second parameter *DataSize* is necessary where the caller needs to specify the size to the data to be processed in bytes. The untyped *Seed* parameter is optional and can be used as cryptographic salt value if the algorithm shall be used for password hashing purposes. If no seed shall be given it is recommended to pass *NullStr* from *SysUtils* there. Since this parameter is untyped it needs a *SeedSize* parameter which specifies the length of the seed passed in bytes. If no seed is given this shall be zero. The last parameter *MaskSize* specifies the length of the output created by this method in byte. The length may be longer or shorter than the length of the *Data* parameter.

The return value is a *TBytes* array of byte.

[6] https://www.ietf.org/rfc/rfc2437.txt

The second overload is like the first one just with the `Data` and `Seed` parameters being `TBytes` arrays thus not requiring the `DataSize` and `SeedSize` parameters. If these shall be used without specifying a seed, it is allowed to pass a zero length `Seed`.

3.4.5.3 KDFx and MGFx

The original author of DEC implemented his own variants of the KDF and MGF algorithms. These are not standardized. The unit test data for those stems from comparing the DEC 6.0 results to DEC V5.2, which of course match.

3.4.5.4 PBKDF2

This is an algorithm for creating a password hash. The caller specifies the password entered, a salt and an iteration count to make it less vulnerable to rainbow table attacks and the desired length of the resulting key in byte. Returned is the generated password hash. For practical purposes it is wise to use a different salt for each password and a high iteration count and store both along with the password hash. When comparing an entered password with the generated hash these parameters are needed again.

3.4.6 Using the cipher algorithms

3.4.6.1 Base structure of the cipher algorithms

The cipher algorithm classes have a mostly common API. Parts of this API are implemented in abstract ancestor classes. The following diagram illustrates this, where abstract classes are written in *cursive*:

```
TDECObject
└─TDECCipher
  └─TDECCipherModes
    └─TDECFormattedCipher
      ├─TCipher_Null
      ├─TCipher_Blowfish
      ├─TCipher_Twofish
      ├─TCipher_IDEA
      ├─TCipher_Cast256
      ├─TCipher_Mars
      ├─TCipher_RC4
      ├─TCipher_RC6
      ├─TCipher_Rijndael
      │ └─TCipher_AES
      │   ├─TCipher_AES128
      │   ├─TCipher_AES192
      │   └─TCipher_AES256
      ├─TCipher_Square
      ├─TCipher_SCOP
      ├─TCipher_Sapphire
      ├─TCipher_DESBase
      │ ├─TCipher_1DES
      │ ├─TCipher_2DES
      │ │ └─TCipher_2DDES
      │ └─TCipher_3DES
      │   ├─TCipher_3DDES
      │   └─TCipher_3TDES
      ├─TCipher_3Way
      ├─TCipher_Cast128
      ├─TCipher_Gost
      ├─TCipher_Misty
      ├─TCipher_NewDES
      ├─TCipher_Q128
      ├─TCipher_RC2
      ├─TCipher_RC5
      ├─TCipher_SAFER
      ├─TCipher_SharkBase
      │ └─TCipher_Shark
      ├─TCipher_Skipjack
      └─TCipher_TEA
        └─TCipher_XTEA
          └─TCipher_TEAN
```

All cipher classes provide all the public methods of `TDECObject` as well as described in chapter *3.4.1 The DEC base class.* The spaces in the class names in the above diagram are in fact _ signs, swallowed by the word processing software due to the small margin between the lines needed to connect the tree lines.

For the ciphers there are two interfaces available:

`IDECCipher` – this is the base interface containing all methods common to all ciphers.

`IDECAuthenticatedCipher` – this interface extends the `IDECCipher` interface with all the additional properties algorithms providing authentication along to the encryption/decryption have. Be aware that while this is implemented by `TDECCipherModes` it will raise exceptions when calling for any cipher mode which does not implement authentication!

> ⚠ In most cases it is not wise to encrypt UTF16 encoded data. For scripts where 8 bits per character are sufficient (e.g. most western ones, if not all), characters in an UTF16 encoded string are stored with 8 bits only. The other 8 bits of the word are 0 and thus known to an attacker, making cracking the encryption easier. Where possible use UTF8 encoded strings instead. For Asian scripts the risk is smaller.

3.4.6.2 TDECCipher

This is the abstract base class for all cipher implementations. Do not create concrete objects from this class! It is being implemented in *DECCipherBase.pas*.

Many of the implemented cipher algorithms are block ciphers, which means that they work on equally sized blocks of data, often on blocks of 8- or 16-byte size. `TDECCipher` only provides abstract methods for encrypting and decrypting a single block of data. The individual cipher classes will override those abstract methods in order to actually provide the encryption/decryption functionality.

You normally do not create instances of this class directly in your code. For encrypting or decrypting data, you will use instances of the concrete cipher classes from the `DECCiphers` unit.

Method	Purpose
`Context`	This class method is inherited from TDECCipher. It returns the characteristics of the encryption algorithm like block size for block-oriented algorithms. Details see in a subchapter following this table.
`Init`	This method must be used to initialize the cipher with the algorithm specific parameters. There exist three overloads of this method so you can pick the one suited best to your data. The parameters which need to be passed are: • The encryption/decryption key. Make sure to select a key with adequate complexity. Simple keys like 1234 or words from dictionaries are unsuitable. Most cipher algorithms also have a minimum and/or maximum key length. • An initialization vector. When you encrypt or decrypt data of a size bigger than the block size of the cipher algorithm each data block is normally mathematically connected with the preceding block. This increases security. The initialization vector is the data needed for the first block, as this one has no preceding block. This also means, that in order to properly decrypt any data you need to know the value of the initialization vector which has been used for encrypting that data. Only the ECB block mode would not need an initialization vector, but this mode should be avoided, as it is inherently less safe! • Filler: this byte value is used to prefill the initialization vector (IV). If an initialization vector is given which is shorter than the required size the IV will contain this defined value. The overloads differ in the data types for the key and initialization vector parameters. Optional one can specify the `PaddingMode` used to fill an incomplete block if a block cipher is being used.
`Done`	This method has to be called after processing the last block of encryption or decryption operation. It properly finalizes the cryptographic operation. If not being called, the cryptographic operation is not complete and you will not process the data of the last block, if a block cipher is being used otherwise the last byte might not have been processed.
`EncodeRawByteString`	This deprecated method encodes string data and returns the encoded data as string. It is only being provided for compatibility

	reasons. The replacement for it is the *EncodeStringToString* method from the *DECFormattedCipher* unit.
DecodeRawByteString	This deprecated method decodes string data and returns the decoded data as string. It is only being provided for compatibility reasons. The replacement for it is the *DecodeStringToString* method from the *DECFormattedCipher* unit.
EncodeBytes	Encodes data passed as a *TBytes* array. The result is a *TBytes* array with the encrypted data. As optional parameter one of the formatting classes can be passed. The formatting will be applied to the encrypted data returned after encryption. For instance, one could return the encrypted data in HEX or BASE64 format.
DecodeBytes	Decodes data passed as a *TBytes* array. The result is a *TBytes* array with the decrypted data. As optional parameter one of the formatting classes can be passed. This would be done in order to remove any formatting applied with passing a formatting class to the *EncodeBytes* method which encrypted the data to be decrypted now.
ClassByName	Searches for a class with the name given as parameter in the class registration list. If a matching class is found, the class reference is returned. This can be used to create an object of that class. So, this method is useful when one wants to create an object of a certain cipher-algorithm implementation but only knows the name of the cipher class as string. If the queried class cannot be found in the registration list an *EDECClassNotRegisteredException* exception will be raised.
ClassByIdentity	If one knows the numeric identify value of a given cipher-implementation class, this method can be used to retrieve the class reference from the registration list. So, this method is useful if one wants to create an object-instance of a certain cipher-implementation class for which one knows the identity value (e.g. because such a value is stored in a file header). If the queried class cannot be found in the registration list an *EDECClassNotRegisteredException* exception will be raised.
IsAuthenticated	Returns true if the algorithm is an authenticated cipher, for example when using the *cmGCM* cipher mode along with a 128 bit block cipher. If true the properties related to authenticated ciphers may be used without getting exceptions for calling unsupported functionality.
CalcMAC	Calculates a message authentication code. This is the encryption of the last block concatenation feedback value. In decryption scenarios it can be used to check if the data arrived unaltered if the sender provides the MAC value along with the encrypted text. Both need to match after decrypting. Using this method is less secure than using the HMAC algorithm!
CalcMACByte	Same as `CalcMAC`, just with `TBytes` as result type.

The class additionally provides these properties:

Property	
InitVectorSize	Returns the size of the buffer for the initialization vector in bytes. The size of this buffer depends on the cipher context of the individual cipher algorithm used.

InitVector	Provides read access to the data of the initialization vector specified as parameter to the *init* method.
Feedback	The data to be encrypted is in most cases bigger than the block size of the used block cipher. In such cases blocks need to be “chained” together to enhance security of the encryption. For this some data of some kind (often derived by a formula or XOR) from the previous block is used as an input parameter for the encryption of the next block. In case of the first block the *InitVector* plays this role. This property provides read-only access to this data.
State	Provides read access to the internal state variable of the cipher. The cipher is implemented as sort of a state machine and with this you can see in which state the cipher operation is, e.g. whether *done* still needs to be called or if it is already initialized by a call to *init* etc.
Mode	Returns the block chaining mode of the cipher and allows to change it. The block chaining mode defines how individual adjacent blocks of cipher data are linked to each other mathematically. It is important to link these blocks in order to strengthen the security of the encryption used. The modes themselves are implemented in *DECCipherModes.pas*, which uses other classes for certain modes.
PaddingMode	Defines how incompletely filled last data blocks will be filled up to their block size. When set to *pmNone* nothing special is being done. When set to *pmPKCS7*, the algorithm from RFC 5652 is being used.

Do not inherit directly from this class if you want to add additional block ciphers, as not using one of the chaining methods from *TDECCipherModes* will result in vulnerable encryption for any data larger than the block size of the algorithm used!

When passing data to *EncodeBytes*, *DecodeBytes*, *EncodeRawByteString* or *DecodeRawByteString* make sure the size of the data passed is the same as the *Context.BlockSize* (or in rare cases *Context.BufferSize*)!

3.4.6.3 TCipherContext

This record is returned by the *Context* class method of each cipher class and provides the basic properties of the cipher algorithm.

Property	
KeySize	Size of the encryption key in byte.
BlockSize	For block-oriented ciphers: size of the block it operates on in byte. For stream-oriented ciphers this will return 1.
BufferSize	Size of the internal processing buffer in byte.
AdditionalBufferSize	Some algorithms use another internal buffer. This is the size of this buffer in byte.
NeedsAdditional BufferBackup	Some algorithms use another internal buffer and some of those who do need it to be saved in some situations.
MinRounds	Minimum value for the *Rounds* property, if the algorithms provides such a property as user changeable value. For all other algorithms (even those having a non-user changeable rounds mechanism) this

	will return 1. This minimum value is enforced in the setter for `Rounds`.
`MaxRounds`	Maximum value for the `Rounds` property, if the algorithms provides such a property as user changeable value. For all other algorithms (even those having a non-user changeable rounds mechanism) this will return 1. This maximum value is enforced in the setter for `Rounds`.
`CipherType`	This set tells whether the algorithm is a block- or stream-oriented one and if it is symmetric or asymmetric. At the time writing DEC does not support asymmetric algorithms. For the Null-Cipher a special value `ctNull` is defined.

3.4.6.4 TDECCipherNull

This is a special "do nothing" cipher, which can be used for general testing purposes.

⚠ Make sure you do not use this in production code which relies on encryption as it will not encrypt your data at all!

3.4.6.5 TDECCipherModes

If you want to encrypt data larger than the block size of the block cipher algorithm used, you need to chain blocks. For this several methods have been developed which normally carry over information from one block to another, so the following blocks depend on their preceding blocks. This is being done to make it harder to crack the encryption. If somebody cracks the encryption of one block, he cannot necessarily decrypt any of the previous blocks. Another necessity is to fill up the last block, if it is not completely filled with data. This happens when your data doesn't match block size. Filling up is often called padding.

Both kinds of operations, padding and block chaining, are implemented in the `TDECCipherModes` class, which is implemented in the *DECCipherModes* unit. You normally do not create instances of this class directly in your code. For encrypting or decrypting data, you will use instances of the concrete cipher classes from the `DECCiphers` unit. Those concrete cipher classes will provide all the methods listed here for encrypting and decrypting data and thus these common methods are described here instead for each cipher class again.

Method	Purpose
`Encode`	This method encrypts an untyped memory block. Parameters are the block to be encrypted, a variable which will contain the encrypted data and the size of that block in byte.
`Decode`	This method decrypts an untyped memory block. Parameters are the block to be decrypted, a variable which will contain the decrypted data and the size of that block in byte.

The following available block chaining modes are in so far special, that they provide authentication along with encryption:

- cmGCM – Galois Counter Mode

For details of the available block chaining modes see chapter 3.4.6.11 Picking the right block chaining method. For details about authenticated modes see next chapter.

3.4.6.6 Authenticated cipher modes

The authenticated cipher modes like `cmGCM` are special. They combine block chaining, like the other available cipher modes, with data authentication. This means that additional authentication data can be passed and an additional authentication result is calculated. After encrypting data this calculated result can be passed to the decrypting party and when after decryption (passing the same authentic-cation data as parameter to the decryption as used when encrypting) the same authentication result value is retrieved, data transmission was untampered. If a different authentication value is obtained, there is some issue with the data! When used without specifying any plain text to encrypt but `DataToAuthenticate` only it can be used as a Message Authentication Code (MAC) and is called GMAC.

Since `cmGCM` requires more code than the other modes, it has been implemented in its own unit `DECCipherModesGCM.pas`.

> ⚠ The `cmGCM` mode can only be used together with a block encryption algorithm with a block size of 128 bit! Trying to set this mode for any algorithm with a different block size results in an `EDECCipherException` exception!
> The maximum size of a message GCM can be used with is 2^{39}-256 bits, that about 65 GB.

For this mode these additional properties and methods are available:

Property/Method	Purpose
`GetStandardAuthentication-TagBitLengths`	Returns an array of bit lengths for the calculated `CalculatedAuthenticationResult` value defined by the standard used. It might be possible to use different bit lengths than returned here, but it is recommended to stick to those defined by the standard.
`DataToAuthenticate`	If data authentication of the GCM mode shall be used, the data to be authenticated is put in this *TBytes* property before starting the encryption or decryption. Even if left empty an authentication value will be calculated, based on the data to be encrypted only. ⚠ Defining data to be authenticated without having `Mode` set to one of the available authenticated modes raises an `EDECCipherException`!
`AuthenticationResultBitLength`	Length in bit, the calculated authentication value shall have. It is legal to specify a length > 0 here without putting any data into `DataToAuthenticate`, as

	there will still be an authentication value be calculated. Defining an authentication value bit length without having `Mode` set to one of the available authenticated modes raises an `EDECCipherException`!
`CalculatedAuthenticationResult`	In this property the calculated authentication will be returned after encryption or decryption is finished. After encryption this value should be transmitted to the receiver so he can use it to pass it to `ExpectedAuthenticationResult` to get the decryption results validated. Reading this value without having `Mode` set to one of the available authenticated modes raises an `EDECCipherException`!
`ExpectedAuthenticationResult`	This optional property is used on decryption to specify the expected authentication result. If it is specified and the `CalculatedAuthenticationResult` calculated during decryption does not match the value specified in this property, an `EDECCipherException` is raised! Be sure to call `Done` after decrypting to make this work, as this contains the code for this check! In case the decrypted data does not match the plain text or the `DataToAuthenticate` is wrong, the decrypted data should normally be discarded as either a transmission error has occurred or somebody tampered with the encrypted data. In deviation to the official GCM standard we do not discard the decryption result in such a case, as this would make it completely impossible to recover still readable parts of otherwise corrupted encrypted files. Just be cautious about using such corrupted data!

All these properties are available in the `IDECAuthenticatedCipher` interface, which is implemented by the `TDECCipherModes` class.

3.4.6.7 TDECCipherFormats

All the methods for encrypting and decrypting data, which do not directly work on blocks of data but on *TStreams*, *strings* or *files,* are added in the `TDECCipherFormats` class. All cipher algorithm classes like `TCipher_AES` inherit from it in order to be able to provide these comfort methods without needing to implement those all over again. When adding further ciphers in form of additional classes they always need to inherit from this class!

If you like to use the good programming habit of programming against interfaces instead of using concrete classes, *TDECCipherFormats* is your candidate as well. This class implements the *IDECCipher* interface, which contains all public methods and properties of *TDECCipherFormats* and additionally the initialization methods in case you need to reinitialize the interface reference during your use of it. This can be used for programming against interfaces most cipher algorithms. There might be rare exceptions where a specific cipher algorithm needs additional properties. These are listed at the end of this chapter.

<table>
<tr><th>Method</th><th>Purpose</th></tr>
<tr><td><code>EncodeBytes</code></td><td>Encrypts the data contained in the TBytes based parameter and returns a TBytes array with the encrypted data.</td></tr>
<tr><td><code>DecodeBytes</code></td><td>Decrypts the data contained in the TBytes based parameter and returns a TBytes array with the decrypted data.</td></tr>
<tr><td><code>EncodeStream</code></td><td>Encodes data provides as a stream. The output will be a stream itself. Streams can be any sort of stream like memory or file streams. The following parameters are being passed:
The source stream containing the data to be encrypted. Ensure that the position of this stream is at the starting position of the data to be encrypted.
The target stream into which the encrypted data will be written. The data will simply be appended.
<code>DataSize</code> specifies how many bytes starting from the current position of the source stream have to be encrypted and put into the destination stream.
<code>Progress</code> is an optional parameter to a callback method. This method is called to enable displaying the progress of the current operation. This callback has the parameters Min, Max and Pos. Pos is the position within the source stream. Min is also the position in the source stream and Max is Min plus the number of bytes to be encrypted.</td></tr>
<tr><td><code>DecodeStream</code></td><td>Decrypts data provided as a stream. The parameters of this method are the same as for <code>EncodeStream</code>.</td></tr>
<tr><td><code>EncodeFile</code></td><td>Encrypts the data of a given file. The data will be read out of the specified source file, get encrypted and written into the specified destination file. Source and destination file may not refer to the same file! In addition to the path and file names of the source and destination files the following parameter is available:
<code>Progress</code> is an optional parameter to a callback method. This method is called to enable displaying the progress of the current operation. This callback has the parameters Min, Max and Pos. Pos is the position within the source stream. Min is also the position in the source stream and Max is Min plus the number of bytes to be encrypted.</td></tr>
<tr><td><code>DecodeFile</code></td><td>Decrypts the data of a given file. This is the counterpart of <code>EncodeFile</code> and thus has the same parameters as this function.</td></tr>
<tr><td><code>EncodeStringToBytes</code></td><td>This method takes a string as input, encrypts it and returns the encrypted data as a TBytes array. There exist four overloads of this method. One expects a UnicodeString (you would just pass a normal Delphi string as UnicodeString is an alias for that one) and the other one a RawByteString.
The other two overloads are only available for the Win32 and Win64 compilers. They work on AnsiString and WideString input and TBytes return values.
In addition to the string to be encrypted you can pass an optional formatting class. The formatting will be applied to the encrypted data, so you can for example return the encrypted data HEX or BASE64 encoded.</td></tr>
<tr><td><code>EncodeStringToString</code></td><td>This method takes a string as input, encrypts it and returns the encrypted data as a string. There exist four overloads of this method. One expects a UnicodeString (you would just pass a normal Delphi</td></tr>
</table>

	string as *UnicodeString* is an alias for that one) and the other one a *RawByteString*. The other two overloads are only available for the Win32 and Win64 compilers. They work on *AnsiString* and *WideString* input and return values. The *string*-based overload returns a *string* and the *RawByteString* one a *RawByteString*. In addition to the string to be encrypted you can pass an optional formatting class. The formatting will be applied to the encrypted data, so you can for example return the encrypted data HEX or BASE64 encoded.
`DecodeStringToBytes`	This method takes a *string* as input, decrypts it and returns the encrypted data as a *TBytes* array. There exist four overloads of this method. One expects a *UnicodeString* (you would just pass a normal Delphi *string* as *UnicodeString* is an alias for that one) and the other one a *RawByteString*. The other two overloads are only available for the Win32 and Win64 compilers. They work on *AnsiString* and *WideString* input and *TBytes* return values. In addition to the string to be decrypted you can pass an optional formatting class. This will be used to remove a formatting on the input data. You can for example remove the formatting applied with the `EncodeStringToBytes` method.
`DecodeStringToString`	This method takes a *string* as input, decrypts it and returns the encrypted data as a *string*. There exist four overloads of this method. One expects a *UnicodeString* (you would just pass a normal Delphi *string* as *UnicodeString* is an alias for that one), the result will be a *string* and the other one a *RawByteString* so the result will be a *RawByteString*. The other two overloads are only available for the Win32 and Win64 compilers. They work on *AnsiString* and *WideString* input and return values. In addition to the string to be decrypted you can pass an optional formatting class. This will be used to remove a formatting on the input data. You can for example remove the formatting applied with the `EncodeStringToBytes` method.

⚠ When using the ECBx block chaining method (which is not recommended!) the size of the data passed to any `Encode` or `Decode` method must be a multiple of `Context.BufferSize` (or in rare cases `Context.BufferSize`)! Otherwise an `EDECCIpherException` may be raised!

TDECFormat

This is the abstract base class for the formatting classes. Many methods in `TDECCipherFormats` provide an optional class reference parameter of this type. It can be used to pass a concrete formatting class to be used in that encoding or decoding method. A description of those format classes can be found in chapter *3.4.2 Using the formatting routines*.

3.4.6.8 TCipher_AES key length remarks

In the original design of DEC, the implementation of the AES cipher was a bit special in the way it implements the different key length variants AES128, AES192 and AES256. Instead of providing individual classes for those different key lengths the *TCipher_AES* class automatically detects the key length and the number of rounds it has to perform on the data, which is a direct property of the key length.

Key length in byte	AES variant
0-16	AES128
17-24	AES192
25-32	AES256

Starting with DEC V6.5 the individual classes `TCipher_AES128`, `TCipher_AES192` and `TCipher_AES256` have been introduced. They explicitly raise an `EDECCipherException` exception when calling the `Init` method with a key too long for the AES variant.

3.4.6.9 List of cipher algorithms with properties not included in the IDECCipher interface

One can still use the interface, but needs to be aware that it will not provide access to these additional properties.

- `TCipher_RC5`, this has an additional `rounds` property
- `TCipher_RC6`, this has an additional `rounds` property
- `TCipher_Rijndael` / `TCipher_AES`, this has an additional `rounds` property
- `TCipher_Cast128`, this has an additional `rounds` property
- `TCipher_SAFER`, this has an additional `rounds` and a `version` property
- `TCipher_TEA`, this has an additional `rounds` property
- `TCipher_XTEA` / `TCipher_TEAN`, these have an additional *rounds* property

3.4.6.10 Cipher implementation

The actual implementations of the ciphers currently provided are in `DECCiphers.pas`. Include this unit in your uses clause and create a concrete instance of one of the cipher classes contained in it to encrypt or decrypt data. If you are free to choose which cipher algorithm to use, be sure to read our comments found in the summary XMLDOC comments, as we try to point out algorithms which are being considered as unsafe nowadays. Such algorithms are only being provided for backwards compatibility.

3.4.6.11 Picking the right block chaining method

The following block chaining methods do exist. Each is shortly being described in order to allow you to pick the most suitable one for your task.

> ⚠ The x-variants of the cipher modes are usually creations of the original author of DEC and these are non-standard implementations. Better avoid those if you can.

Block mode	Description
cmCTSx	Double CBC, with CFS8 padding (filling up) of a not completely filled last block
cmCBCx	Cipher Block Chaining, with CFB8 padding (filling up) of a not completely filled last block. Each plain text block is being XORed with the preceding block before it gets encrypted. The first block is being XORed with the init vector. It is wise to use a new value for the init vector for each encryption you do and the method is not really suited for situations where single bytes arrive which do not fill a complete block yet, as it has to wait until a block is full before it can start.
cmCFB8	8-bit cipher feedback mode. This mode works with a shifting register. The content of this register depends on the whole history of the plain text fed to the cipher algorithm. Reoccurring plain text in a data stream thus always gets encrypted differently. If there is a transmission error in one bit it affects as many bits as the shifting register contains. They will be incorrectly decrypted.
cmCFBx	Cipher feedback mode, but on the block size of the cipher used
cmOFB8	8 bit output feedback mode
cmOFBx	Output feedback mode, but on the block size of the cipher used
cmCFS8	8 bit CFS with double CFB
cmCFSx	Like CFS, but on the block size of the cipher used
cmECBx	DECs implementation of the electronic code book algorithm. Since this does not chain blocks together at all you should avoid it if possible! This is the least secure mode!
cmCTS3	This one is only available if you enable the *`DEC3_CMCTS`* define in *`DECOptions.inc`*. It is being provided for compatibility reasons with old DEC versions only. Do not use it in new code!
cmGCM	Galois counter mode: this one provides an additional authentication feature, which can be used to easily check that the data has not been tampered with between encryption and decryption. For details see chapter *3.4.6.6 Authenticated cipher modes*.

3.4.6.12 Interfaces for the cipher classes

If you like to use the good programming habit of programming against interfaces instead of using concrete classes you can do so. There are several interfaces for different aspects/kinds of ciphers provided.

Interface	Implemented by	Purpose
IDECCipher	*TDECFormattedCipher*	Contains all public methods and properties of *TDECFormattedCipher*, except for the ones contained in *IDECAuthenticatedCipher*. This interface can be used for programming against interfaces with all cipher algorithms. When using an authenticated cipher mode, it is just not sufficient as it lacks the additional properties required for those.
IDECAuthenticatedCipher	*TDECCipherModes*	Contains all properties needed for using the authentication features of authenticated block chaining modes. Encryption/Decryption would be done via the *IDECCipher* interface.

3.4.7 Using the random number generators

The random number generator provides pseudo random numbers (Delphi's built in Random function would provide pseudo random numbers as well, as nearly all random number generators in ordinary computers can only provide pseudo random numbers unless specialized hardware is available) and is written in a not object-oriented way. It is suited especially for cryptographic purposes.

The *DECRandom.pas* unit contains two different generator algorithms. By default, the better (but slower) one using a hash algorithm is being used. The default hash algorithm used is SHA256 but it can be changed by assigning a different class to the global variable `RandomClass`. If the weaker but faster algorithm shall be used the `DoRandomBuffer` global variable needs to be set to *nil*. As this is a global variable one can even provide his own random number generator implementation for applications which already use *DECRandom.pas* if desired.

If the *AUTO_PRNG* define is defined, which is the default setting in *DECCoptions.inc*, the random number generator is initialized automatically in the initialization section of *DECRandom.pas*. In that case it is initialized with the current system time. If the define is not set the random number generator must be manually initialized before first use by calling one of the two overloads of *RandomSeed*. The parameter-less one initializes with the current system time, the other one accepts parameters with initialization data.

> If the random number generator is not specifically initialized a repeatable deterministic generator is the result. This results in always getting the exact same random number sequence, which should be avoided!

<table>
<tr><th>Procedure/Function</th><th>Purpose</th></tr>
<tr><td><code>RandomSeed</code></td><td>There are two overloads available for this procedure. If the defaults are kept, the parameter less one initializes a non-repeatable random number generator with a seed value (start value) based on RandomSystemTime. If the parameterized one is used the seed value initialized depends on the value of the Size parameter and if applicable on the contents of the Buffer parameter.
<table>
<tr><td>Size = 0</td><td>The initial seed value is set to 0 and the generator is repeatable. This should be avoided!</td></tr>
<tr><td>Size > 0</td><td>The generator is repeatable but initialization is based on Buffer contents as well. So, if that one contains random data the seed value will be random.</td></tr>
<tr><td>Size < 0</td><td>The seed value is based on RandomSystemTime. This is less random than specifying a really random Buffer with Size > 0 but better than the Size = 0 case!</td></tr>
</table></td></tr>
<tr><td><code>RandomLong</code></td><td>Returns an unsigned 32 bit random number</td></tr>
<tr><td><code>RandomBuffer</code></td><td>This procedure needs to have an already created buffer passed in and as 2nd parameter the number of random bytes to create. The passed buffer is being filled with the number of random bytes specified, starting from the first byte of this buffer.</td></tr>
<tr><td><code>RandomBytes</code></td><td>Returns a TBytes array filled with random bytes. The number of bytes to be returned is specified with the <code>Size</code> parameter.</td></tr>
<tr><td><code>RandomRawByteString</code></td><td>Returns a string filled with random data. The size of the string in byte</td></tr>
</table>

	is specified with the *Size* parameter. This procedure is deprecated and we recommend to use the *RandomBytes* function instead.
RandomSystemTime	This function creates a seed value for random number generation based on the system time and on *QueryPerformanceCounter* (up to Delphi 2010 and Windows only) or based on the system time and *TStopWatch.GetTimeStamp*, which is available on all platforms.

3.4.8 Useful helper routines

The helper routines described in this chapter are to be found in the `DECUtils` unit.

Procedure/Function	Description
`ReverseBits`	Reverses the bits in the parameter passed and returns them as return value. Passing 10111111111111110000000000000000 results in 00000000000000001111111111111101.
`SwapBytes`	Swaps the order of bytes of the passed in parameter. A parameter containing 01 02 03 04 hexadecimal will be returned as 04 03 02 01. The buffer passed in will contain the swapped values after the call. As 2nd parameter the size of the buffer to be swapped in bytes needs to be passed.
SwapUInt32	Swaps the order of bytes of the passed in parameter. A parameter containing 01 02 03 04 hexadecimal will be returned as 04 03 02 01. In this case it is a function with the swapped *UInt32* as return value.
SwapUInt32Buffer	This method gets an untyped source buffer and an untyped destination buffer passed. Both buffers will be treated as arrays of *UInt32* values and both buffers need to be either of the same size or the destination buffer needs to be bigger than the source buffer. The bytes of the *UInt32* values in the source buffer will be swapped each then be placed into the destination buffer. The order of the *UInt32* values stays the same, but the bytes in them will have been swapped. The parameter `Count` specifies the number of *UInt32* values contained in the source parameter.
SwapInt64	Swaps the order of bytes of the passed in parameter. This function is the same as `SwapUInt32`, just for *Int64* typed data. The sign bit is not being specially treated.
SwapInt64Buffer	This method is similar to *SwapUInt32Buffer* just with *UInt64* data elements instead of *UInt32* ones. The sign bit is not being specially treated.
XORBuffers	Connects the bytes of two buffers passed by XOR each. You have to pass two buffers with the bytes that shall be XOR-connected, a size parameter for specification of the buffer size passed in byte and an output buffer. Both input buffers and the output buffer need to have at least a size as specified with the `Size` parameter.
ProtectBuffer	Securely overwrites the untyped buffer passed as parameter. Additionally, to the buffer the buffer size in bytes needs to be passed as parameter.
ProtectStream	Securely overwrites the contents of a stream. Starting from the current position within the stream `SizeToProtect` bytes will be securely overwritten. You may pass in any stream type.
ProtectBytes	Securely overwrites all the bytes of a passed in `TBytes` array of bytes.
ProtectString	This procedure exists in four overloads. It securely overwrites all the bytes in a *string, RawByteString, AnsiString* or *WideString*. The latter two types are only available for the Win32 and Win64 compilers
BytesToRawString	Creates a *RawByteString* out of the bytes in a *TBytes* array. The bytes will be put into the string as is, means if such a byte contains a value of $41 the resulting character of the string will be “A”. No special provisions are being made for control characters or characters outside the 7 bit ASCII range. Use this procedure with care!
BytesToRawString	Converts a byte array to a RawByteString. An empty array will result in an empty string.
RawStringToBytes	Converts a RawByteString to a byte array. An empty string will result in an empty array.

BytesToString	Converts a byte array to a string using unicode encoding. An empty array will result in an empty string.
StringToBytes	Converts a string to a byte array using unicode encoding. An empty string will result in an empty array.
IsEqual	Compares the contents of two *TBytes* buffers for equality

3.4.9 TDECProgressEvent – displaying progress of an operation

How can progress be displayed during a lengthy encryption/decryption or hashing operation?

In `DECTypes` there is a type `TDECProgressEvent`. This is a reference to an anonymous method and because of this it can be used in conjunction with normal methods, regular procedures and with inline anonymous method code.

The *TStream* and file-based methods contain an optional parameter of this event type. If you implement it either pass a method or a normal procedure containing the same parameters as defined in `TDECProgressEvent` or write the in-place code for an anonymous method.

If you use this event the event handler passed will be called in these situations:

- Directly before beginning of the operation. You will get the number of bytes to process by this. `State` will be `Started` at this point. When called for this case `Pos` will always be zero.
- Each time a chunk of data has been processed. You will get the position by this. `State` will be `Processing` at this point.
- Directly after finishing the operation, which is when the finalize block is executed. If an exception is raised during processing the finalize block will be reached as well after exception handling and the finished event will be called as it normally would be called anyway. `State` for this finishing event is `Finished`. When the event is called for this case, `Pos` will always be the same as `Max`.

The event has three parameters:

`Max`	Number of bytes to be processed. In case of a file this will always be the file size. In case of a stream it will be the size passed as parameter to the stream processing method.
`Pos`	This is the position of the operation relative to the starting position. In case of a file this will be relative to the start of the file and in case of a stream this will be relative to the position the stream was at when the stream processing method has been called.
`State`	This gives the reason why the progress event has been called.

3.4.10 DECOptions.inc

The `DECOptions.inc` include file contains a few global defines which influence how DEC works. Most of those should be left alone as they are needed to proper function of DEC on different platforms.

If you want to disable some define simply put a . between the { and the $.
Example: `{.$DEFINE NO_ASM}`
To enable a disabled define simply remove the . between { and $.

Those defines which may be enabled or disabled without problems are in the section titled “User configuration”. These specifically are:

- `{$DEFINE AUTO_PRNG}`, when used DEC always uses his own pseudo random number generator instead of the Delphi standard *random* function.

- `{$DEFINE NO_ASM}`, when used none of the assembler versions of the routines are used. Only pure Pascal implementations are used then. If you want to use DEC on a non Win32 platform this define needs to be on! On Win32 disabling the define can give you some smaller speed gains.

- `{.$DEFINE DEC52_IDENTITY}`, when used this DEC version uses the same identity identifier value DEC 5.2 used. }This enables to read files created with DEC V5.2 which used that identity identifier.

- `{.$DEFINE DEC3_CMCTS}`, when enabled the CTS3 block cipher mode is made available. It is not recommended to be used, since it is a less secure mode! This option is only there for cases where one needs to deal with data which has been encoded with the `cmCTS` mode of DEC V3.0.

- `{.$DEFINE FMXTranslateableExceptions}`, enable this if you intend to use DEC in a Firemonkey mobile project and want to be able to translate the exception messages without needing to capture the exceptions.

- `{$UNDEF OLD_SHA_NAME}`, enable this if you like to use the old class name for the SHA0 hash class. For clarity the `THASH_SHA` class got renamed to `THASH_SHA0` in DEC 6.0.

- `{$UNDEF OLD_WHIRLPOOL_NAMES}`, enable this if you like to use the old class names for the Whirlpool hash classes.

- `{.$DEFINE ManualRegisterHashClasses}`, enable this if you do not want to have all hash- classes automatically registered in the initialization sections of the `DECHash` unit. The same exists for `DECFormat` and `DECCiphers` units in form of the `{.$DEFINE ManualRegisterFormatClasses}` and `{.$DEFINE ManualRegisterHashClasses}` defines. If you want to use the class registration mechanism in such a case, you need to manually register those hash-, format- or cipher-classes you want to use with the mechanism.

- `{$UNDEF OLD_REGISTER_FAULTY_CIPHERS}`, enable this if you want to have the cipher class variants with faulty implementations, as they were implemented in DEC V5.2 and thus provided for backwards compatibility only, registered in the class registration mechanism (if automatic registration has been turned on by you) as well.

3.4.11 Translating exception messages

By default, all exception messages used by DEC have been declared as resource strings, containing English text.

On Win32/Win64 resource strings are stored in special tables inside the generated exe-file automatically and most application translation tools are able to pick them up and provide some mechanism for translating those. This works equally well for VCL and for Firemonkey (FMX) applications.

Firemonkey on the other hand doesn't support this scheme on mobile platforms. On those resource strings do compile but are treated as normal string constants. Translation tools are not able to replace them, unless the places where they are being used (e.g. displayed on screen) are wrapped into a call of the *Translate* function from *FMX.Types*.

In order to fix this, the `FMXTranslateableExceptions` define must be enabled. This enables special constructors for the `EDECException` class and its descendants. Those will use the defined resource strings but feed them to the FMX *Translate* function before assigning them to the exception class.

Your translation tool still might not identify those texts (some do) as it would be complicated for it to follow your source, but they usually allow to manually add texts to be translated. The output of such tools will be a *.lng file* usually, which you load into a *TLang* component you place on your main form. That component will provide all texts to your components and for the translate function of *FMX.Types*.

3.4.12 List of no longer recommended algorithms

The following algorithms are no longer recommended for use due to security issues. They are still contained in DEC for compatibility reasons and "the sake of completeness":

Ciphers:

- DES, is considered to be too weak nowadays[7].
- NewDES, it can be broken too easily.
- Skipjack, is considered to be too weak nowadays[8].
- 3Way, it is vulnerable to differential cryptoanalysis
- Square, as there exists a specialized attack on this one
- IDEA, as there exist classes of weak keys and some other successful attacks[9].
- TEA, as it is known that three other equivalent keys exist for each key and because of other existing attacks[10].
- XTEA_DEC52, as this is the faulty implementation of the XTEA algorithm as it was contained in DEC V5.2. It is being provided for backwards compatibility only in case you have old data encrypted with that implementation.
- SCOP_DEC52, as this is the faulty implementation of the SCOP algorithm as it was contained in DEC V5.2. It is being provided for backwards compatibility only in case you have old data encrypted with that implementation.

Hashes[11]:

- MD2, is considered to be broken at least on paper.
- MD4, is considered to be broken at least on paper.
- MD5, is considered to be broken (collisions): HMAC using MD5 is still considered to be ok[12].
- SHA0, has known issues with the initialization[13].
- SHA1, is considered to be broken (collisions). HMAC using SHA1 is still considered to be ok[14].
- SHA224, is considered to be too weak by German BSI[15].
- HAVAL-128, collisions have been found.
- RIPEMD, but only the original variant.
- RIPEMD128, because the message digest length of 128 bit is considered to be too small[16]. All other RIPEMD variants are still considered to be ok.
- PANAMA

[7] https://en.wikipedia.org/wiki/Data_Encryption_Standard
[8] https://en.wikipedia.org/wiki/Skipjack_(cipher)
[9] https://en.wikipedia.org/wiki/International_Data_Encryption_Algorithm
[10] https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm
[11] https://en.wikipedia.org/wiki/Hash_function_security_summary
[12] https://en.wikipedia.org/wiki/HMAC
[13] https://en.wikipedia.org/wiki/SHA-1
[14] https://en.wikipedia.org/wiki/HMAC
[15] https://de.wikipedia.org/wiki/SHA-2
[16] https://en.wikipedia.org/wiki/RIPEMD

- Tiger, at least the 192 Bit variant (which is the only one currently implemented in DEC) is considered to be broken, at least on paper.
- Whirlpool0, newer variants are ok but the Whirlpool1 variant is the recommended one as it is safer than WhirlpoolT.[17]
- Keccak: this one is only provided for compatibility reasons. For new code SHA3 shall be used as this is the same algorithm except for NIST having added two bits as some sort of Algorithm ID.

3.4.13 Updating code which used DEC 5.x

Ok, between DEC 5.x and 6.0 quite a few structural changes were made. Some to enhance maintainability of the library and some to be able to support other platforms.

Here are some relevant points for moving code forward:

- DEC 5.x had a data type named `binary`. This was an alias for `AnsiString`. This does no longer exist. It is not recommended to use strings as buffer for binary data, as convenient as this was back then. In XE7 arrays learned some new tricks thus implementing some of the convenience found in strings. While some new methods using `TBytes` based buffers as parameters were defined those having `binary` parameters got changed to using `RawByteString`. That should keep compatibility as much as possible while still being cross platform compatible.
- Most units got renamed, mostly where name parts were formerly abbreviated.
- New units got introduced to separate functionality. For the Hash, Cipher and Format classes there is a base unit now: `TDECHashBase`, `TDECCipherBase` and `TDECFormatBase`. All concrete implementations of algorithms inherit from the base class defined in that one.
- The concrete algorithm implementations are in the `TDECHash`, `TDECCiphers` and `TDECFormat` units.
- The class diagrams found in the preceding chapters might provide further overview to the new structure.
- Another source of information is the `VersionHistory.pdf` file!

[17] https://en.wikipedia.org/wiki/Whirlpool_(hash_function)

3.5 The class registration mechanism

The classes *TDECHash*, *TDECCipher* and *TDECFormat* do contain a registration mechanism where all descendant classes are registered as meta-classes in a generic list. This mechanism is helpful when you build an application which shall contain a list of algorithms to pick from, so you can dynamically list the available algorithms and create instances of those. All those classes inherit this mechanism from the *DECBaseClass* unit, where it is implemented in *TDECClassList*. The FMX GUI demo applications show their use.

Each of the formatting, cipher or hash classes is being registered into the appropriate class list in the initialization section of the und implementing the particular class. The class list is implemented as a generic *TDictionary* and provided as a public *class var* of the base class of the formatting, cipher or hash classes.

⚠ Whether the formatting, hash or cipher classes are automatically registered (and thus their code is compiled in!) is controlled by some defines in *DECOptions.inc*. By default, these defines are on and thus all such classes are registered.

Each class type is registered with a property called *identity* as key. This identity is a unique *Int64* number specifying the class. For instance, you may store this number in the header of some encrypted file to record with which algorithm it was encrypted. With the class registration mechanism, you can easily find the right class used for deciphering the file and create the necessary instance of this cipher class. Besides the ability to loop through all registered class types in the list, the mechanism provides two methods for searching a class type reference:

- *ClassByName* – searches for a given long or short class name. Examples: *TDECFormat_HEXL* is a long name or *HEXL* would be the short name. If such a class is registered in that list the class reference will be returned and you can call the *Create* constructor on this to create an object reference of this type returned. If no class with such a name is registered an *EDECClassNot-RegisteredException* exception is being thrown.
- *ClassByIdentity* – searches for a given unique ID. If a class with the given *Identity* is registered in that list the class reference will be returned and you can call the *Create* constructor on this to create an object reference of this type returned. If no class with such a name is registered an *EDECClassNotRegisteredException* exception is being thrown.
- *GetClassList* – with this method you can get a string list of all the classes registered. Just pass any valid *TStrings* or *TStringList* object as parameter and you will have the long names of all the registered classes.

Another extension of this mechanism was implemented when the first password hash class was implemented. All password hash classes shall inherit from the *TDECPasswordHash* class. This class introduces a second registration/finding mechanism, which can be used at least for some password hash implementations like BCrypt.
Those algorithms which are supported as Crypt/BSD password hashes implement a protected class method called *GetCryptID* which returns the ID Crypt/BSD use for this algorithm in their password record storage. In order to get a class reference by searching for it with its Crypt/BSD ID the

ClassByCryptIdentity class method has been added. It searches for a class within the list of registered hash classes with the specified Crypt/BSD ID and returns a *TDECPasswordHashClass* class reference if it finds one. If no class with the specified ID is found in the list an *EDECClassNotRegisteredException* exception will be raised. The returned class reference can be used to create an object instance of that class and perform the necessary password calculation/checking calls on that.

Example:

```
Uses
  Generics.Collections, DECHashBase;

var
  MyClassRef : TPair<Int64, TDECClass>;
  Identity : Int64;
begin
  Identity := 123;
  If TDECHash.ClassList.TryGetValue(Identity, MyClassRef) then
    ShowMessage(MyClassRef.Value.ClassName);
end;
```

If you like to search for a class reference by its *ClassName*, you can use the *ClassByName* class function of the corresponding base class.

Example for finding a class reference and creating an object instance from it:

```
Uses
  DECHashBase;

var
  Hash:TDECHash;
begin
  Hash := TDECHash.ClassByName('THash_MD5').Create;
  try
    Hash.Init;
  finaly
    Hash.Free;
  end;
end;
```

The class type list mechanism allows for registering and unregistering new classes at runtime and it is implemented in such a way that if the DEC Unit implementing a registered class type is being unloaded because it belongs to a package which is being unloaded, the class type will be unregistered. This prevents you from retrieving class references from a registration list of classes which are no longer available. You cannot try to create an object reference from it and cause an access violation because the class implementation is no longer available.

3.6 Unit Test TestInsight integration

DEC's DUnit unit test project has integrated support for Stefan Glienke's TestInsight IDE plugin. Using this plugin, the unit tests can be run in background without interrupting one's workflow. Depending on settings they are run in fixed intervals or every time the project is saved. Alternatively, they still can be run manually.

How to set this up?

1. Download the TestInsight installer. It is linked in this wiki page: https://bitbucket.org/sglienke/testinsight/wiki/Home

2. Close the IDE and install it.

3. Start the IDE and open the DEC project group.

4. Call View/TestInsight Explorer. A window will pop up. This window can be docked somewhere in the IDE. The Object Inspector might be a good place, because it is not used when working in the DUnit test project or use the right half of the messages panel.

5. Activate the DUnit test project.

6. Right click on the DUnit test project and select TestInsight from the context menu. This enables use of TestInsight for running the tests instead of the DUnit GUI.

7. In TestInsight panel either click on the disk button to run the tests every time the project is saved or on the timer button to run the tests in a fixed interval.

8. If the disk button has been selected save the unit test project. TestInsight will create a list of all available tests and run them. By default, tests are listed by status, but it might be more helpful to select the *list by fixture* option, as this resembles the same grouping of tests as shown in DUnit GUI, which might be more familiar and might be easier for finding a particular test, especially given that DEC already contains over 1,000 tests.

3.7 Extending DEC

This chapter describes what to consider when adding new formatting, cipher or hash classes to DEC. If you do extend DEC in any way it would also be nice if you would send us your source code modification so we can add it to the next release, if deemed useful for the general audience of DEC! Of course, we will mention you in the DEC hall of fame: the list of contributors!

And remember: whatever you add needs to have unit tests implemented by you!

> ⚠ If you add a new formatting class, a new hash class or a new cipher class do not forget to register it via the RegisterClass class procedure as otherwise the demo applications will not automatically pick it up.

3.7.1 Structure and style

If adding or modifying anything it would be really nice and helpful to stick to a certain style and structure. If the modification will flow back into the main repository/project this will make things easier. Here a list of things to consider:

1. When adding new units do add the copyright notice, as found in already existing units, at the top of the unit.
2. Do not use syntax or libraries not supported since at least the minimum Delphi version DEC currently claims support for! This minimum version is specified in chapter 1.
3. For Delphi we want to use unit namespace syntax in the *uses* sections.
4. DEC tries to be FPC compatible, but that cannot deal with unit namespaces yet. Use the proper IFDEFs, as seen in the already existing units, to make it work with both compilers.
5. In implementation source code do not use unit namespace syntax, as this would not be FPC compatible.
6. There might be things which cannot be made FPC compatible at all. If something like that is required put it in appropriate IFDEF sections so FPC does not “see” it.
7. After changing or adding something try to update or add unit tests for it.
8. When creating a pull request for something consider these rules:
    a. The pull request should contain a single commit only, if possible.
    b. The pull request should be focussed on a single topic and the topic should not be too broad. Better split up large topics into several smaller pull requests and before starting a large topic better start a discussion with the maintainers before to avoid too differing views on the topic and the pull request to be rejected.
    We do like additional participants, but better discuss things first before implementing them.
    c. Try to describe the contents of the pull request and where applicable the aim. When adding a new cipher for instance the aim is clear but when changing some method, it might not be clear why it shall be changed.
    d. When turning in a pull request be open for discussion about its contents and about possible requests to modify something in it.
    e. Do not turn in any modifications/additions which do not compile!

3.7.2 Adding new ciphers

New cipher classes added to DEC should always descend from *TDECFormattedCipher* from the *DECCipherFormats* unit. They need to provide at least implementations for the following methods, from *TDECCipher* from the *DECCipherBase* unit. This means they need to be overwritten:

- *DoInit*
- *DoEncode*
- *DoDecode*

While you can overwrite the ***Encode***, ***Decode*** and the protected *EncodeXXX/DecodeXXX* methods from *TDECCipherModes* you normally do not need to. This would be rather uncommon!

Register your algorithm by adding a *TCipher_XXX.RegisterClass(TDECCipher.ClassList);* call to the implementation section of the *DECCiphers* unit. Without doing so your class will not appear in any demo which makes use of the registration mechanism.

After implementing your new cipher class, it is good practice to implement the basic set of unit tests for it as well. Get at least one reliable set of input data and corresponding encrypted data. Reformat the encrypted data to be in the *TFormat_ESCAPE* format as this is the standard for our unit tests. Then look at the existing unit tests in the *TestDECCipher* unit and implement such tests for your new cipher class.

> ⚠ If you add some new cipher algorithm we would like to know about it and it would be nice if you could share it with us so it becomes part of the standard version of DEC.

3.7.3 Adding new cipher block concatenation modes

If you like to add a new cipher block chaining mode you need to add the following methods to the *TDECCipherModes* class in the *DECCipherModes* unit:

- *EncodeXXX*
- *DecodeXXX*

XXX is the name of your mode.

Additionally, the *TCipherMode* enumeration in the *DECCipherBase* unit needs your mode added as a new value and then you need to update the *Encode* and *Decode* methods of the *TDECCipherModes* class in the *DECCipherModes* unit. You need to add your new enumeration value to the case statement and call the *EncodeXXX* or *DecodeXXX* methods for your new mode.

After adding your mode make sure it works by adding some unit tests. For this add a `TestEncodeXXX` and `TestDecodeXXX` method to the `TestTDECCipherModes` class in the `TestDECCipherModes` unit. Make sure you have valid test data from a trustable source to do so.

> ⚠ If you add some new cipher padding algorithm we would like to know about it and it would be nice if you could share it with us, so it becomes part of the standard version of DEC.

3.7.4 Adding new cipher padding algorithms

Padding algorithms are being used for block ciphers only to fill up any incomplete last block. For example, if the block cipher used has a block size of 8 bytes, but somebody wants to encrypt 25 bytes there are 7 bytes which are unused in the last block. The padding mode chosen defines how these 7 bytes are filled.

If one wants to add a new padding mode he has to perform the following steps:

- Create a new *TXXXPadding* class in *DECipherPaddings.pas* which descends from *TPadding*. Implement all methods defined in *TPadding*. XXX stands for the name of the padding algorithm added.
- Add another enumeration value to *TPaddingMode* in *DECCipherBase.pas*.
- Add the necessary calls to the methods of your newly created TXXXPadding class in the methods of *TDECFormattedCipher* in *DECCipherFormats.pas*. For this look for any uses of *FPaddingMode* in that unit.
- Add unit tests for your new padding class in *TestDECCipherPaddings.pas*. Check that they are passed.

> ⚠ If you add some padding algorithm we would like to know about it and it would be nice if you could share it with us, so it becomes part of the standard version of DEC.

3.7.5 Adding new hash algorithms

If you like to add a new hash algorithm add a new class *THash_XXX* to the *DECHash* unit where XXX is the name of your algorithm. Your class needs to override at least *DoTransform*, *DoInit*, *Digest*, *DigestSize* and *BlockSize* methods. Digest is needed to return the actual calculated hash value as different hash algorithms usually have different hash sizes and thus differ in the internal FDigest field definition which is thus not defined in the *TDECHashBase*, *TDECHashAuthentication*, *TDECHashExtended* or *TDECHashPassword* base classes. For Merkle-Darmgard based algorithms is can often be found in the *THashBaseMD4* class, as most, if not all of them, either stem from this one directly or indirectly inherit from it. If your new algorithm uses that Merkle-Darmgard approach it should inherit from *THashBaseMD4* as well and that already inherits from *TDECHashExtended*.

Your new class usually should descend from the *TDECHashExtended* class, unless it is a hash algorithm specifically designed for hashing passwords. In that case it should inherit from

TDECPasswordHash to avoid having *CalcStream* and *CalcFile* methods which are usually not relevant for password hashing and to have a *Salt* property already defined, which is needed by most password hash algorithms.

Register your algorithm by adding a *THash_XXX.RegisterClass(TDECHash.ClassList);* call to the implementation section of the *DECHash* unit. Without doing so your class will not appear in any demo which makes use of the registration mechanism.

Now it is time to add unit tests. Fetch good test data from a reputable source and add a unit test class to the *TestDECHash* unit similar to this one (XXX is the name of your hash algorithm):

```
// Test methods for class THash_XXX
{$IFDEF DUnitX} [TestFixture] {$ENDIF}
TestTHash_XXX = class(THash_TestBase)
public
  procedure SetUp; override;
published
  procedure TestDigestSize;
  procedure TestBlockSize;
  procedure TestIsPasswordHash;
  procedure TestClassByName;
  procedure TestIdentity;
end;
```

Fill in the methods. Look the necessary contents up in one of the other test classes. Adapt your test data. For getting the identity of your class you might want to run your new unit tests. The test for the identity will fail as you did not yet adapt your identity test value. Note the value your test calculated and change the expected value to that one.

⚠ If you add some new hash algorithm we would like to know about it and it would be nice if you could share it with us, so it becomes part of the standard version of DEC.

3.7.6 Adding new password hash algorithms

In order to add a new password hash algorithm simply follow the advice given in the preceding chapter. If your algorithm is one supported by Crypt/BSD you should override the following methods in addition:

Method	Purpose
class function GetCryptID:string	Returns the Crypt/BSD unique ID assigned to the algorithm. Enables retrieval of the algorithm via *ClassByCryptIdentity*.
function GetCryptHash(Password : TBytes; const Params : string; const Salt : TBytes;	This returns a Crypt/BSD style password hash storage string as stored in a Crypt/BSD password "database" text file. The overload with the *Password* para-

`Format        : TDECFormatClass): string`	meter declared as a string calls this one internally so that automatically works.
`function IsValidPassword( const Password   : string; const CryptData : string; Format         : TDECFormatClass): Boolean`	Check whether a give password belongs to the given Crypt/BSD password storage string. The overload with the *Password* para-meter declared as a string calls this one internally so that automatically works.

Unit tests should be added as well of course! In case of such password hash classes the unit test class should inherit from the *THash_TestPasswordBase* class. A good example of which tests to implement for a password hash class provided by the *TestTHash_BCrypt* class in *TestDECHash.pas*.

Reference implementations can be found in the *THash_BCrypt* class. Algorithm specific parameters like special cost factors should be added as properties if necessary.

⚠ If you add some new password hash algorithm we would like to know about it and it would be nice if you could share it with us, so it becomes part of the standard version of DEC.

3.7.7 Adding new formatting classes

In order to add a new formatting a class with the following signature usually needs to be added to the *DECFormat* unit. In some rare cases the class looks a bit different, an example for this would be the *TFormat_Radix64* class. Make sure your class only contains class methods or class vars but no regular methods or fields!

```
/// <summary>
///   Description of your new format
/// </summary>
TFormat_XXX = class(TDECFormat)
protected
  class procedure DoEncode(const Source; var Dest: TBytes;
                           Size: Integer); override;
  class procedure DoDecode(const Source; var Dest: TBytes;
                           Size: Integer); override;
  class function DoIsValid(const Data;
                           Size: Integer): Boolean; override;
public
  class function CharTableBinary: TBytes;
  class function FilterChars: string;
end;
```

Implement all those class methods.

Register your algorithm by adding a *TFormat_XXX.RegisterClass(TDECFormat.ClassList);* call to the implementation section of the *DECFormat* unit. Without doing so your class will not appear in any demo which makes use of the registration mechanism.

Now it is time to add unit tests. Fetch good test data from a reputable source and add a unit test class to the *TestDECFormat* unit. For this look at the already implemented test classes, copy the signature of the one fitting best and insert this under a new name matching your new format's name. Then implement all the test methods the same way the methods for the already existing class have been implemented.

⚠ If you add some new formatting algorithm we would like to know about it and it would be nice if you could share it with us, so it becomes part of the standard version of DEC.

⚠ When adding a new formatting class make sure it only contains class functions / class procedures and class vars. Otherwise some places where your class is being used in DEC might not function, as DEC expects not to work on object instances of these formatting classes but on the class itself via class methods.

3.7.8 Adding new CRC variants

⚠ If you add some new CRC polynomial we would like to know about it and it would be nice if you could share it with us, so it becomes part of the standard version of DEC.

Please ensure you have valid test data for the new CRC variant you would like to add before actually doing so. Just adding new variants without proper unit tests does not help anybody.

Adding a new CRC variant requires to add a new enumeration value to the `TCRCType` type in `DECCRC.pas`. The enumeration value should be added at the end. It further requires adding an entry to the `CRCTab` constant. The entry should be added at the end of the table. The entry consists of the polynomial value, the number of bits the CRC operates on, the start value with which the CRC is to be initialized, the initialization value for the finalization vector and a Boolean value defining whether the polynomial is an inverse one.

After adding the necessary definitions to the `DECCRC` unit you need to add a unit test for it. To do so open the `TestDECCRC` unit and add the following new published methods *TestCRCInitCRCXXX* and *TestCRCXXX* where XXX is the name of your new CRC. A private *SetUpCRCXXX* method is usually required as well.

3.7.9 Adding unit tests

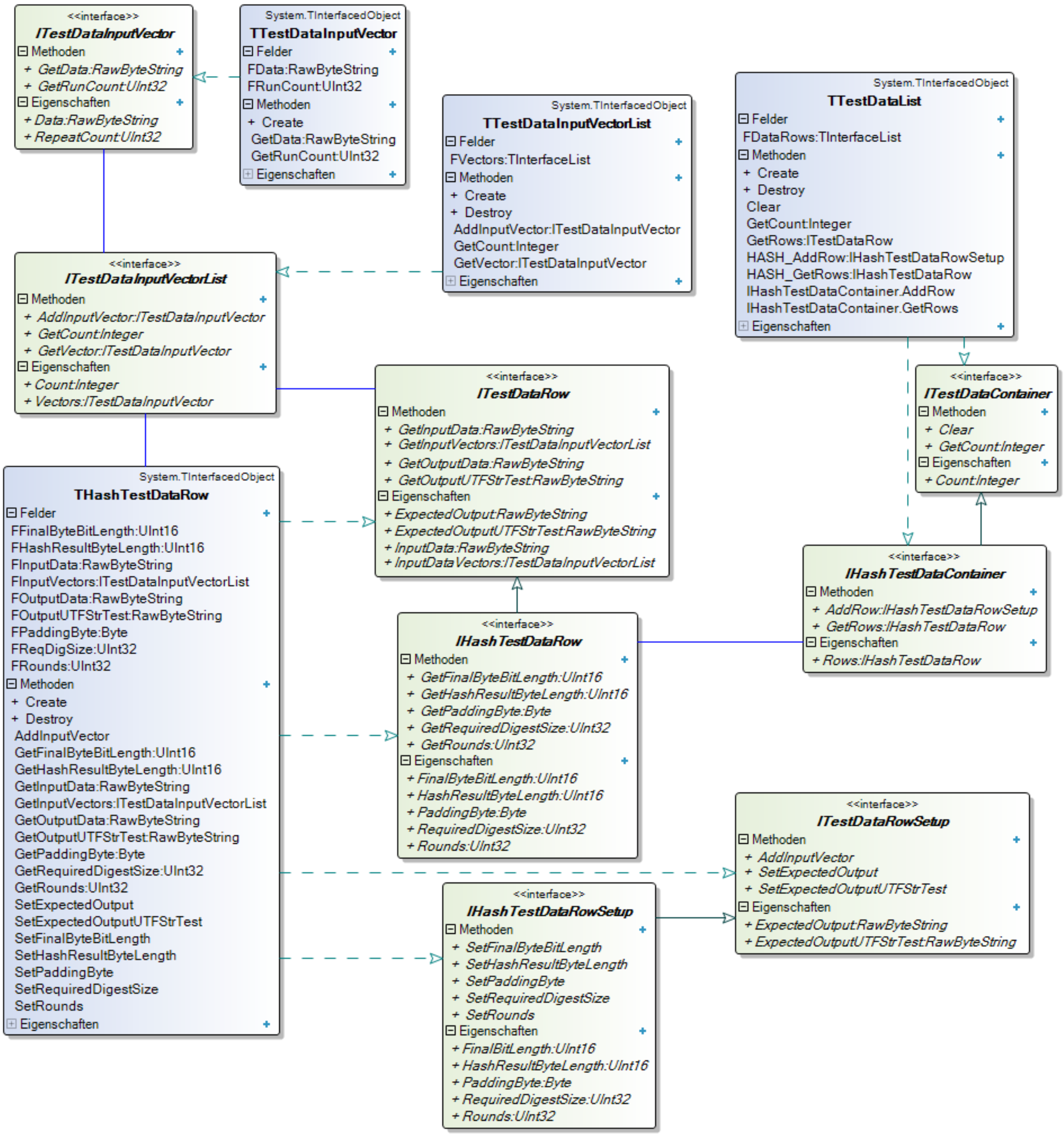
There are unit tests available for nearly all methods etc. shipping in the default DEC distribution, except for some of the random number functions or some of their behaviour. These unit tests have been written in a way that they can be run as *DUnit* tests and as *DUnitX* tests as well. This has been done because *DUnit* still has the better GUI test runner, while *DUnitX* tests can basically be run on other platforms as well.

For this there are two unit test projects provided. One for *DUnit* and one for *DUnitX*. If you want to switch between those you need to either define or undefine the *DUnitX* define in *defines.inc* of the unit tests. Otherwise you will get compilation and/or runtime errors.

The unit test implementation units are in the *Unit Tests\Tests* subfolder. Look at the tests already implemented, sometimes you just need to add further test data as many tests work on arrays/structures of test data. In other cases, you may want to add new test methods. Whatever you do: please let us know! We may add your enhancements to DEC so all users will profit from expanded test coverage!

3.7.10 Hash unit test data management

In an attempt to unify the handling of unit test data the following architecture has been created. It is currently being used for the hash unit tests only. Here's a class diagram:



Some things to note:

1. The interfaces have been set up as an inheritance hierarchy as ciphers most likely will need a few changes (if they should use this architecture one day) so there is a common base interface.
2. There is a strict differentiation between interfaces returning test data and interfaces setting up test data.
3. The `InputData` contained in `ITestDataRow` can be a concatenation of several `AddInputVector` calls.

4 Demos

In order to make your life easier, DEC ships with some demo applications. This chapter lists them and their purpose.

- Cipher_Console
  Simplistic demo showing how to encrypt and decrypt some string. If new to the topic of encryption start here.

- Cipher_Console_KDF
  Variant of the simple demo, showing how to improve security of the encryption by hashing the key used for encrypting and decrypting using a key deviation function.

- Cipher_FMX
  This cross platform compatible demo, using the Firemonkey GUI framework, allows the user to choose a cipher algorithm, a cipher block chaining mode and a format conversion class. The user can encrypt or decrypt a string he enters afterwards. When cipher mode cmGCM is selected additional GCM specific properties can be set and authentication is being used.

  The demo is way more advanced than the `Cipher_Console` demo as it demos the class registration mechanism as well.

- Format_Console
  Simplistic demo for showing how to use one of the formatting classes to change the format of a given string.

- Hash_Console
  Simplistic demo showing how to calculate a hash value over a given string. If new to hash value calculation start here.

- Hash_FMX
  This cross platform compatible demo, using the Firemonkey GUI framework, allows the user to select a hash algorithm and format conversion classes for the input and the output data. The user can enter some text to be hashed then. The text will first get formatted with the input format class, the hash value will be calculated and then the output formatting will be applied before displaying the output. With a checkbox the user can enable a live output mode where the output is updated after each entered character.

  The demo is way more advanced than the `Hash_Console` demo, as this one shows the use of the class registration mechanism as well and for algorithms with specific properties those are mostly provided as well.

- Password_Console
  A simple console based demo showing all password related methods using BCrypt as algorithm.

- HashBenchmark
  A simple FMX based benchmarking application for all hash algorithms. It calculates how long it takes to hash 10 MB of data for each algorithm and based on the time stopped via `TStopwatch` the MB/s speed is calculated and displayed. For password hashes the size of the data, over which the hash is calculated, is automatically reduced to the MaxPasswordLength specified by the algorithm and the number of calculation rounds is automatically adapted.

- Random_Console
  A simplistic demo showing how to use the random number generator.

- RandomComparison_VCL
  A simplistic demo to compare DEC's pseudo random number generator with that of the RTL. It shows the distribution of random numbers in the range 0-255 for both DEC's generator and that of the RTL in a TChart each.

- ProgressDemoVCL
  A simple VCL based demo for encrypting a file and displaying the progress while encrypting. One can select the method for progress display so all three possible ways are demoed.