CmpCrypto and CmpX509Cert

The source code of the examples can be found inside the `CryptoDemo.project` for downloading.

Cryptographic methods are important to achieve the following goals when processing data:

- **Integrity** of data will ensure that the recipient can be confident that the data can not be modified unnoticed.
- **Authenticity** means that the recipient can be sufficiently certain that a record has really been created by its supposed author.
- **Confidentiality** aims to prevent unauthorized access to the relevant data under all circumstances.

We distinguish between three cases in which the terms mentioned above play a major role:

- This is data that is currently being processed on the local processor (**Data in Use**).
  In this case, data security depends very much on the design and properties of the local hardware. Since the data has to be processed at this moment, active encryption, for example, is not useful. The hardware environment must, for example, ensure that the frequency patterns resulting from the processing of the data cannot be traced back to the original data content.

- This is data that is transported from one location to another (**Data in Motion**).
  The data is transported via so-called transport channels. One channel may be protected against manipulation and monitoring by third parties. We then talk about a safe channel. Without these safety features, we are talking about an unsafe channel. For transmission through an unsafe channel, the data packets can be prepared by encryption and a signature. This protects them against interception and unnoticed changes. Encrypting and signing data usually involves a lot of computing effort.When using a secure channel, additional data handling should be avoided because the channel already provides all necessary security measures and the additional effort by signing or encrypting makes no sense.

- This is data stored in a specific location (**Data at Rest**).
  If a site for storing our data is protected against unnoticed manipulation of our data, then we are talking about a trusted location. If these assumptions do not apply to a particular location, we are talking about an untrusted location. By encrypting our data and creating a signature, data can be stored in untrusted locations without the risk of unauthorized use or unnoticed modification.

The collection of functions in the CODESYS libraries `CmpCrypto.library` and `CmpX509Cert.library` significantly reduces your own effort required to enable secure data processing. But the functionality provided requires proper use. These functions are an important basis for secure data processing. But the security of the data and its processing also depends on the environment in which the library functionality is used. It depends very much on how the results of these functions are linked and whether the correct function has been selected for the task in hand.

Cryptographic protection is usually based on two basic assumptions:

- Good, long random numbers can be generated efficiently.
  The quality of random numbers depends on the quality of the random number generator, which is unpredictable in its output and does not give preference to the generation of certain numbers. The large length of a random number is important because it increases the effort to guess the number by simply trying it out.

- There are functions whose effort to calculate their inverse function differs greatly from the effort to calculate the actual function. We then speak of so-called one-way functions.
  In a broader sense, functions are also referred as one-way functions for which no reversal is known to date, that is practically calculable within a reasonable period of time.

  The last sentence reveals a risk of all currently used cryptographic procedures. The more powerful the next generation of computers becomes, the more likely it is that a specific inversion function can be calculated in acceptable time. This means that it is no longer possible to talk about a one-way function and the related algorithm has become vulnerable.

Note

Example of a One-Way function:
The multiplication of two prime numbers can be “easily” computed, while for the inversion, the prime factorization, currently no efficient algorithm was published... (see: Prime decomposition)

Therefore, it is an extremely important prerequisite for secure data processing that the algorithms used can be changed at any time and without delay if it becomes known at a certain point in time that the currently used algorithm can be successfully attacked and thus its protective function is no longer guaranteed.

Note

It is a good idea to design your own functions in such a way that the algorithms used for certain cryptography functions can be easily replaced. The different algorithms available in the CODESYS environment are encoded in the `RtsCryptoID` enumeration. The important parameters such as “block size” or “key size” of a certain algorithm must be determined e. g. with the help of Wikipedia. After CDS-58920 is available the neccecary information can be queried with the `CryptoGetAlgorithmInfo` library function.

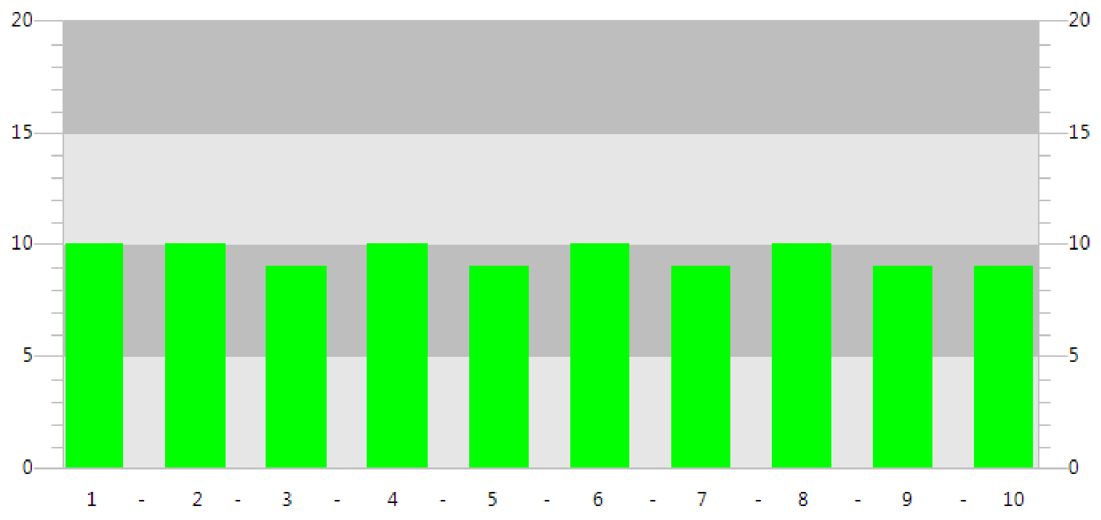
References:

- Performance
- Key exchange
- RSA Kryptosystem
- Hybride_VerschlÃ¼sselung
- Difference sha-1, sha-2, sha-256 hash algorithms/
- Advanced Encryption Standard
- Cryptographic Message Syntax (CMS)

Random Numbers

The following code example produces a series of random integers in the range 1 to 10:

```
VAR
    udiCounter : UDINT;
    usiNumber : USINT;
    bsNumber : RtsByteString := (ui32MaxLen:=SIZEOF(usiNumber), pByData:=ADR(usiNumber));
    Result : RTS_IEC_RESULT;
END_VAR

Result := CryptoGenerateRandomNumber(ui32NumOfRandomBytes:=SIZEOF(usiNumber), pRandom:=ADR(bsNumber
Random := (usiNumber MOD 10) + 1;
```



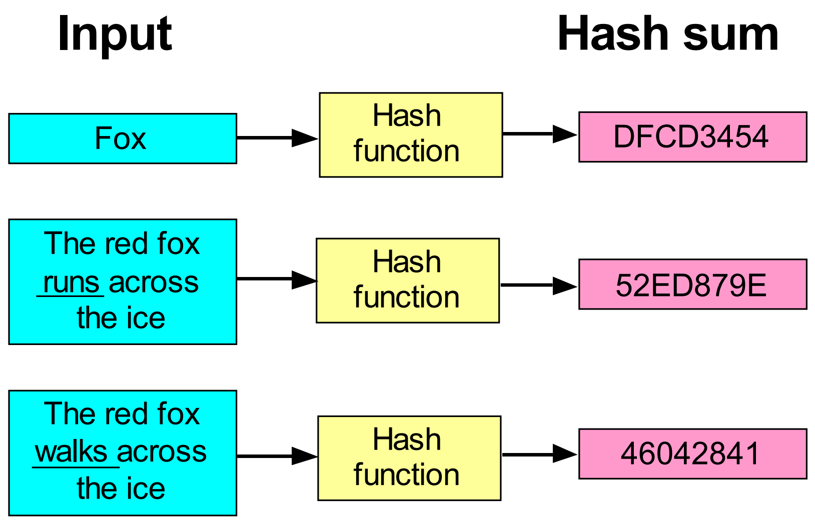
After some time, the histogram shows an equal distribution of the generated random numbers (in the example of numbers 1 to 10). This is one of the expected quality criterion for a random number generator. Another criterion for the quality of random numbers is variance of the initial values. The harder it is to predict the generated series of numbers, the more secure are the algorithms for encryption based on this random numbers. (See: Wikipedia)

See:

- https://www.random.org/
- **Analysis of an On-line Random Number Generator**, April 2001, Louise Foley
- **Random Number Generators**, April 2005, Charmaine Kenny

Hashing

A hash function can map the content of a message (a memory area) of any length to a byte array with a fixed length. This property is often used to determine whether a particular message has been changed, for example, during transport to the recipient. In addition to the way in which messages are exchanged, the sender and receiver must also have agreed on which hash algorithm they want to use. The recipient can then apply the same hash function to the received message and compare the result with the received hash value. The message was most likely not modified if the two hash values are identical. The hash value of a message is often called its fingerprint. Using a fingerprint, the consistency of the data transmitted can be detected with a high probability. However, the authenticity of the transmission cannot be determined. It is very easy to imagine that on the way from the sender to the recipient the message can be intercepted, modified and a new correct hash code can be calculated and forwarded to the recipient.



A typical hash function at work. Note that the hash sums are always the same size, no matter how short or long the input is. Also note that the hash sums look very different even when there are only slight differences in the input. The hash sums seen here (in hexadecimal format) are actually the first four bytes of the SHA-1 hash sums of those text examples. (See: Wikipedia)

For example, to check a message with the SHA1 algorithm, the following actions are necessary:

- Provide a memory area for the message.
- Prepare an additional memory area and initializes it with the related hash code.
- The SHA1 algorithm calculates a hash code with a length of 20 bytes. Therefore, this amount of memory area must be appropriately provided.

```
VAR
    _hHASH : RTS_IEC_HANDLE := CryptoGetAlgorithmById(ui32CryptoID:=RtsCryptoID.HASH_SHA1, pResult:
    sMessage : MESSAGE := 'The red fox runs across the ice';
    abyHashCode : HASH_CODE := [
        16#52, 16#ED, 16#87, 16#9E, 16#70,
        16#F7, 16#1D, 16#92, 16#6E, 16#B6,
        16#95, 16#70, 16#08, 16#E0, 16#3C,
        16#E4, 16#CA, 16#69, 16#45, 16#D3
    ];
    xMessageOK : BOOL;
END_VAR
```

```
xMessageOK := CheckMessage(sMessage, abyHash);
```

The algorithm has in CODESYS the id `RtsCryptoID.HASH_SHA1`. With this assumption, a method for checking the message could look like this:

```
METHOD CheckMessage : BOOL
VAR_INPUT
    sMessage : REFERENCE TO MESSAGE;
    abyHashCode : REFERENCE TO HASH_CODE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    bsMessage : RtsByteString := (ui32MaxLen:=SIZEOF(sMessage), pbyData:=ADR(sMessage), ui32Len:=TO
    bsHashCode : RtsByteString := (ui32MaxLen:=SIZEOF(abyHashCode), pbyData:=ADR(abyHashCode), ui32
    abyNewHashCode : HASH_CODE;
    bsNewHashCode : RtsByteString := (ui32MaxLen:=SIZEOF(HASH_CODE), pbyData:=ADR(abyNewHashCode),
    diCmpResult : DINT;
END_VAR
```

```
Result := CryptoGenerateHash(hAlgo:=_hHASH, pData:=ADR(bsMessage), pHash:=ADR(bsNewHashCode));
diCmpResult := SysMemCmp(pBuffer1:=ADR(abyHashCode), pBuffer2:=ADR(abyNewHashCode), udiCount:=SIZEO
CheckMessage := diCmpResult = 0;
```

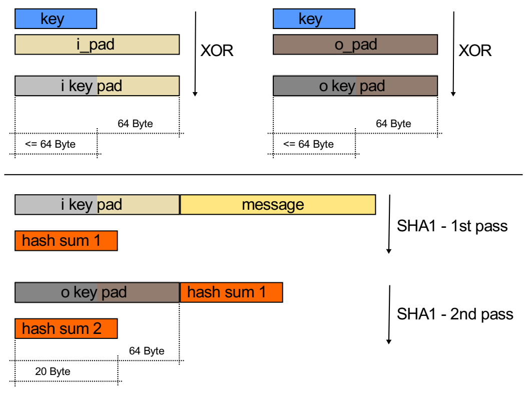
Note

The comparison of hash values must be carried out very carefully. A seemingly successful comparison leads to a message, for which the hash values have been calculated, being mistakenly accepted as unchanged!

Calculating hash values is a good thing if you are looking for an efficient method to convert large amounts of data into a relatively short byte sequence. This byte sequence is very likely to uniquely identify the content of the data set, if the hashing algorithm is selected appropriately. For example, only relatively short hash values can be used to compare large amounts of data. A special feature of good hashing algorithms in general is important. Two different data sets are very likely to never provide the same hash value (Collision-Resistant Hash Function).

Hashed Based Message Authentication Code

A hash-based message authentication code (HMAC) can be used to sign a record. In this way, HMAC can be used at a later date to ensure that the content of the data has not been falsified in the meantime and that the identity of the data source has not changed. The difference to the simple application of a hash algorithm from the last chapter is that this method includes a secret key in addition to the data. Therefore, in addition to the knowledge of the selected algorithm and the actual data, a secret key is also required to check the HMAC.



Description of the SHA-1 HMAC Generation. See: Wikipedia

In the following example, an agreement was made regarding a secret (`_sSecret`) and the hash algorithm to be used (`HMAC_SHA1`). On this basis, messages can now be signed (`SignMessage`). The signature obtained from this procedure can then be used at a later point in time to check the integrity of the message (`VerifyMessage`).

```
VAR
    _hHMAC : RTS_IEC_HANDLE := CryptoGetAlgorithmById(ui32CryptoID:=RtsCryptoID.HMAC_SHA1, pResult:
    _sSecret : SECRET := 'MySecret';

    abySignature : SIGNATURE;
    xMessageOK : BOOL;
    sMessage : MESSAGE := 'The red fox runs across the ice';
END_VAR
```

```
abySignature := SignMessage(sMessage);
xMessageOK := VerifyMessage(sMessage, abySignature);
```

The same secret is required for generating the signature and checking it. As long as the secret remains really secret, it is very unlikely that someone can change the message content unnoticed and create a suitable signature. This probability depends on the strength of the secret, of course. The easier this secret can be guessed (Rainbow Table), or the faster it can be found through repeated testing (Brute-force Attack), the lower the achievable security against unwanted manipulations.

```
METHOD SignMessage : SIGNATURE
VAR_INPUT
    sMessage : REFERENCE TO MESSAGE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    bsSecret : RtsByteString := (ui32MaxLen:=SIZEOF(_sSecret), ui32Len:=TO_UDINT(LEN(_sSecret)), pB
    ksStorage : RtsCryptoKeyStorage := (byteString:=bsSecret);
    ckSecret : RtsCryptoKey := (keyType:=RtsCryptoKeyType.KeyType_Key, key:=ksStorage);
    bsMessage : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(LEN(sMessage)), pB
    bsSignature : RtsByteString := (ui32MaxLen:=SIZEOF(SIGNATURE), ui32Len:=0, pByData:=ADR(SignMes
END_VAR
```

```
Result := CryptoHMACSign(
    hAlgo:=_hHMAC,
    pData:=ADR(bsMessage),
    key:=ckSecret,
    pSignature:=ADR(bsSignature)
);
```

```
METHOD VerifyMessage : BOOL
VAR_INPUT
    sMessage : REFERENCE TO MESSAGE;
    abySignature : REFERENCE TO MESSAGE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    bsSecret : RtsByteString := (ui32MaxLen:=SIZEOF(_sSecret), ui32Len:=TO_UDINT(LEN(_sSecret)), pB
    ksStorage : RtsCryptoKeyStorage := (byteString:=bsSecret);
    ckSecret : RtsCryptoKey := (keyType:=RtsCryptoKeyType.KeyType_Key, key:=ksStorage);
    bsMessage : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(LEN(sMessage)), pB
    abyNewSignature : SIGNATURE;
    bsSignature : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=0, pByData:=ADR(abyNewSig
    diCmpResult : DINT;
END_VAR
```

```
Result := CryptoHMACSign(
    hAlgo:=_hHMAC,
    pData:=ADR(bsMessage),
    key:=ckSecret,
    pSignature:=ADR(bsSignature)
);
diCmpResult := SysMemCmp(pBuffer1:=ADR(abySignature), pBuffer2:=ADR(abyNewSignature), udiCount:=SIZ
VerifyMessage := diCmpResult = 0;
```

Note

The comparison of signature values must be implemented very carefully. A seemingly successful comparison leads to a message being mistakenly accepted as unchanged!

Encryption

By encrypting, a “plain text”, i. e. a clearly readable text, is converted into a “cipher text”, i. e. into an obscure character string. The terms plain text and cipher text have evolved historically and can be seen symbolically. In addition to text messages, it is also possible to encrypt other types of information such as voice messages, image recordings or the source code of programs. The underlying cryptographic principles remain the same.

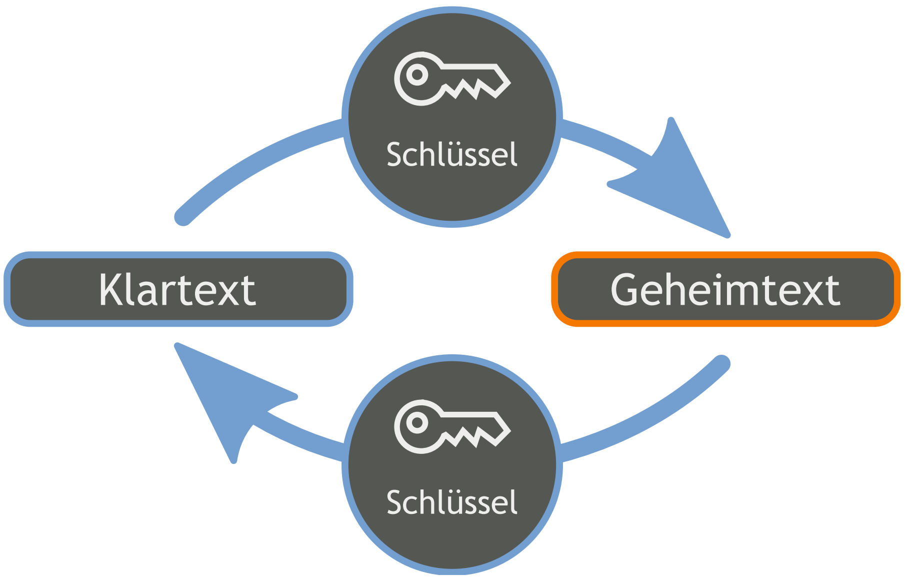
Symmetric Encryption

Encryption methods that work with a secret key that is used both for encryption and decryption are called symmetric methods and they are part of the area of symmetric cryptography. Almost all symmetrical encryption algorithms are optimized for restricted environments. They are characterized by low hardware requirements, low energy consumption and are easy to implement in hardware.

The encryption methods of symmetric cryptography work with a single key, which must be present during encryption and decryption. These procedures are fast and with correspondingly long keys, they also offer a high level of security.

The crucial point is the key exchange between the communication partners. Before a encrypted data transmission can start, the communication partners must agree on a specific key and exchange it. If an attacker has the key, he can not only decrypt the data, but also encrypt the data himself without the respective communication partner is being able to noticing it. A secure key exchange is a central problem of symmetric cryptography. This problem can be solved by methods of asymmetric encryption.

The use of asymmetric cryptography for key exchange and the use of symmetric cryptography for the transport of messages provides security against the loss of the secret keys and utilises the speed advantage that symmetric methods usually offer. Please refer to Hybrid Encryption or the “Pay by Use” Example for more detailed information.

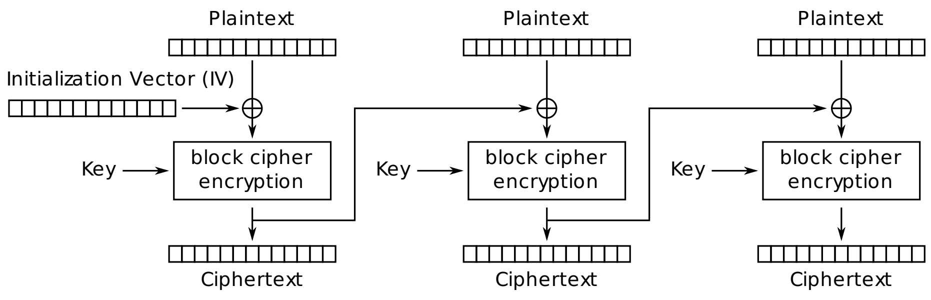


Exemplary usage of AES-256 in CBC Mode

Further References:

- Advanced Encryption Standard (AES)
- Cipher Block Chaining (CBC)
- Initialization vector
- Padding

```
VAR
    _hCipher : RTS_IEC_HANDLE := CryptoGetAlgorithmById(ui32CryptoID:=RtsCryptoID.AES_256_CBC, pRes
    _szBlock : ULINT := 16; // Blocksize of ``_hCipher`` => 128 Bit for AES-256-CBC
    _sKey : SECRET; // 256 Bit Secret Key
    _szKey : ULINT := 32; // 256 Bit Key length
    _sInitVector : MESSAGE; // Random Initial Value of Length ``_szBlock``
END_VAR
```



Cipher Block Chaining (CBC) mode encryption

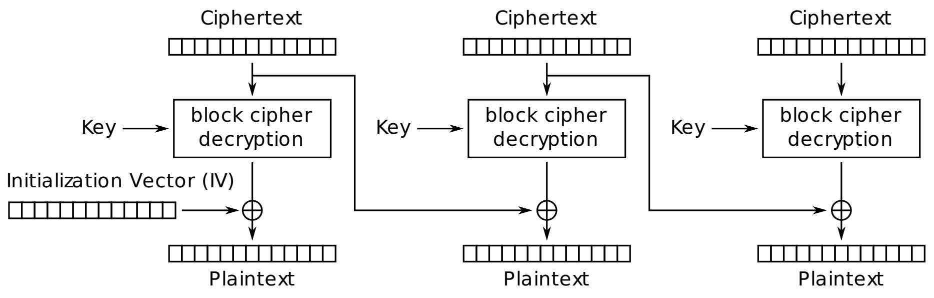
Using a randomized init vector and reusing the cipher text from the precursor stage each cipher text block depends on all plain text blocks processed up to that point. With this each cipher text block is unique. So it is not possible to get the same cipher text block for a repeated content of a plain text block. The key is a common secret between the party who encrypts the message end the party who decrypts the message. The init vector data has to be the same on both sides, but these data has not to be kept secret.

```
METHOD EncryptMessage : ULINT
VAR_INPUT
    sPlainText : REFERENCE TO MESSAGE;
    szPlainText : ULINT;
    sCipherText : REFERENCE TO MESSAGE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;

    bsKey : RtsByteString := (ui32MaxLen:=SIZEOF(_sKey), ui32Len:=TO_UDINT(_szKey), pByData:=ADR(_s
    ksStorage : RtsCryptoKeyStorage := (byteString:=bsKey);
    ckKey : RtsCryptoKey := (keyType:=RtsCryptoKeyType.KeyType_Key, key:=ksStorage);

    bsInitVector : RtsByteString := (ui32MaxLen:=SIZEOF(_sInitVector), ui32Len:=TO_UDINT(_szBlock),
    bsPlainText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(szPlainText), pB
    bsCipherText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=0, pByData:=ADR(sCipherT
END_VAR
```

The randomized initialization of the key and the init vector makes it very difficult to predict the results of the encryption process.

```
Result := CryptoGenerateRandomNumber(ui32NumOfRandomBytes:=TO_UDINT(_szKey), pRandom:=ADR(bsKey));
Result := CryptoGenerateRandomNumber(ui32NumOfRandomBytes:=TO_UDINT(_szBlock), pRandom:=ADR(bsInitV
```

```
Result := CryptoSymmetricEncrypt(
    hAlgo:=_hCipher,
    pPlainText:=ADR(bsPlainText),
    key:=ckKey,
    pInitVector:=ADR(bsInitVector),
    xEnablePadding:=TRUE,
    pCipherText:=ADR(bsCipherText)
);
IF Result = ERR_OK THEN
    EncryptMessage := bsCipherText.ui32Len;
END_IF
```



Cipher Block Chaining (CBC) mode decryption

```
METHOD DecryptMessage : ULINT
VAR_INPUT
    sCipherText : REFERENCE TO MESSAGE;
    szCipherText : ULINT;
    sPlainText : REFERENCE TO MESSAGE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;

    bsKey : RtsByteString := (ui32MaxLen:=SIZEOF(_sKey), ui32Len:=TO_UDINT(_szKey), pByData:=ADR(_s
    ksStorage : RtsCryptoKeyStorage := (byteString:=bsKey);
    ckKey : RtsCryptoKey := (keyType:=RtsCryptoKeyType.KeyType_Key, key:=ksStorage);

    bsInitVector : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(_szBlock), pByD
    bsCipherText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(szCipherText),
    bsPlainText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=0, pByData:=ADR(sPlainTex
END_VAR
```

```
Result := CryptoSymmetricDecrypt(
    hAlgo:=_hCipher,
    pCipherText:=ADR(bsCipherText),
    key:=ckKey,
    pInitVector:=ADR(bsInitVector),
    xEnablePadding:=TRUE,
    pPlainText:=ADR(bsPlainText)
);
IF Result = 0 THEN
    DecryptMessage := bsPlainText.ui32Len;
END_IF
```

A easy procedure for testing the two methods from above is to compare the original message (`abyPlainText`) with the decrypted message (`abyDecryptedText`).

```
szCipherText := EncryptMessage(abyPlainText, TO_ULINT(LEN(abyPlainText)), abyCipherText);
szDecryptedText := DecryptMessage(abyCipherText, szCipherText, abyDecryptedText);
IF szDecryptedText > 0 THEN
    diCmpResult := SysMemCmp(pBuffer1:=ADR(abyPlainText), pBuffer2:=ADR(abyDecryptedText), udiCount
END_IF
```

Note

An effective and thus secure encryption is not only ensured by the fact that someone selects e.g. the AES algorithm. It depends very much on the appropriate selection of the parameters “Key length”,”Key value” and “Operating mode”. If these parameters are not selected correctly, an attacker may find it very easy to get to know the supposedly well hidden secret.

Whenever possible, an already existing encryption mechanism should be used. For example, the use of the TLS protocol is usually preferred for encrypted communication between two computers via a TCP/IP network. The data can then be transmitted over such a secure channel without the application having to worry about encryption itself.

Combination of Signing and Encrypting

As described in Hashed Based Message Authentication Code, these methods can be used to determine the integrity and originator of a message beyond doubt. In this context, it is important to follow a strict sequence of procedures. In a first step, the message should be encrypted only in the second step the HMAC should be generated. By this procedure, the receiver can check in a first step by the evaluation of the HMAC whether the received message was changed and then decrypt only an unmodified message. It is important that messages with an invalid HMAC are not decrypted but discarded unseen. The result of the decryption process with specially prepared messages can provide an attacker with important information that allows him to draw conclusions about the parameters of the encryption method.

Asymmetric Encryption

In order to be able to use the methods for asymmetric encryption, the sender of a message creates a key pair beforehand. A “private key” and a “public key” are created. The private key must be kept secret. It is not possible to draw conclusions about the private key from the public key, so it is completely safe to make the public key generally accessible.

The key pair can now be used to perform various operations:

- The public key can be used by anyone to encrypt data records. However, decryption of this data can only succeed with the help of the private key.
- The private key can be used to sign a data record. With the help of the public key this signature can be verified and so the originator of the data and the integrity of these data can be determined without doubt.

Thus, if the sender and receiver of a message manage their own key pair, all tasks of an encrypted communication can be accomplished without the two partners having to exchange a common, secret information. It is thus ensured that the recipient of a message can ascertain the originator of this message without doubt and can check at any time whether the message was falsified on its way to him. Both participants can be sure that it was not possible for an attacker to gain knowledge of the content of the message.

That sounds like a perfect method for the secure exchange of data. But there are a few important things to consider:

- The construction of the key pairs must follow a number of rules. It is therefore necessary to use a trustworthy software and to keep it up to date with the latest state of the art. Without these measures, an attacker may be able to leverage the integrity, authenticity or security of the data.

- The private key must never come to the attention of anyone other than its owner.
- The owner of a public key must be clearly identifiable. This is usually done via a certificate issued by a trusted authority. It is therefore very important to verify the certificate before using the related public key.
- The operations of asymmetric encryption are very time consuming and are not suitable for handling a large amount of data.

The use of asymmetric cryptography for the key exchange only (small amount of data) and the use of symmetric cryptography for the transport of messages (large amount of data) utilities the speed advantage that symmetric methods usually offer. This combination provides a solution of the key exchange problem in the field of symmetric cryptography.

Please refer to Hybrid Encryption or the “Pay by Use” Example for more detailed information.

Key pair Generation

In the CODESYS environment, the PLC shell and the security screen provide many tools for generating, managing, importing and exporting certificates and keys. The following illustrations are based on the use of these tools.

In order to create a key pair on a PLC, a pseudo component is required that can serve as a client for the certificate store of the CODESYS runtime system.

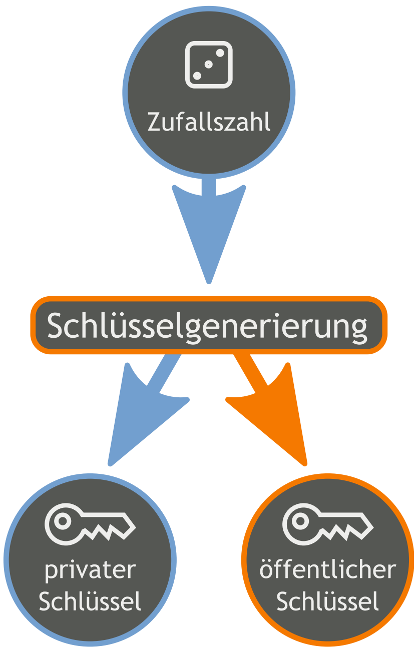
```
hComponent := CMAddComponent(sComponentName, dwComponentId, dwComponentVersion, ADR(Result));
IF Result = ERR_OK THEN
    hCertStore := X509CertStoreOpen(dwComponentId, ADR(Result));
END_IF
```

The new component can be identified in the list of all certificate store client components by its name (e.g. “MyTestApplication”):

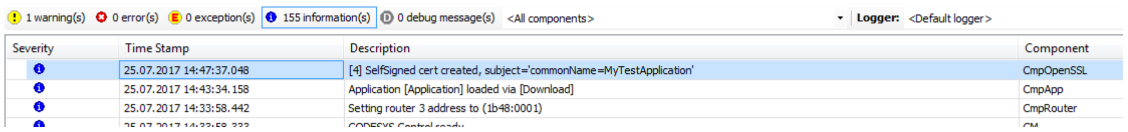
```
> cert-getapplist

Nr.    ComponentName              CommonName                       CertAvailable DateNotBefore
------------------------------------------------------------------------------------------------
0      CmpOPCUAServer             OPCUAServer@DOLLWONB             FALSE        --
1      CmpSecureChannel           DOLLWONB                         FALSE        --
2      CmpApp                     DOLLWONB                         FALSE        --
3      CmpWebServer               DOLLWONB                         FALSE        --
4      MyTestApplication          MyTestApplication                FALSE        --
```

In the next step, a self-signed certificate is created for the component specified by the index parameter with a valid period of time of a certain duration.

```
> cert-genselfsigned 4 expdays=10

Generate self-signed certificate with given index with valid time of 10 day(s). Check logger to see
```



Checking the logger will result in an similar picture like the following:



1 warning(s) 0 error(s) 0 exception(s) 155 information(s) 0 debug message(s) <All components> Logger: <Default logger>

Severity	Time Stamp	Description	Component
●	25.07.2017 14:47:37.048	[4] SelfSigned cert created, subject='commonName=MyTestApplication'	CmpOpenSSL
●	25.07.2017 14:43:34.158	Application [Application] loaded via [Download]	CmpApp
●	25.07.2017 14:33:58.442	Setting router 3 address to (1b48:0001)	CmpRouter

The list of certificates now available can be viewed using the "cert-getcertlist" command:

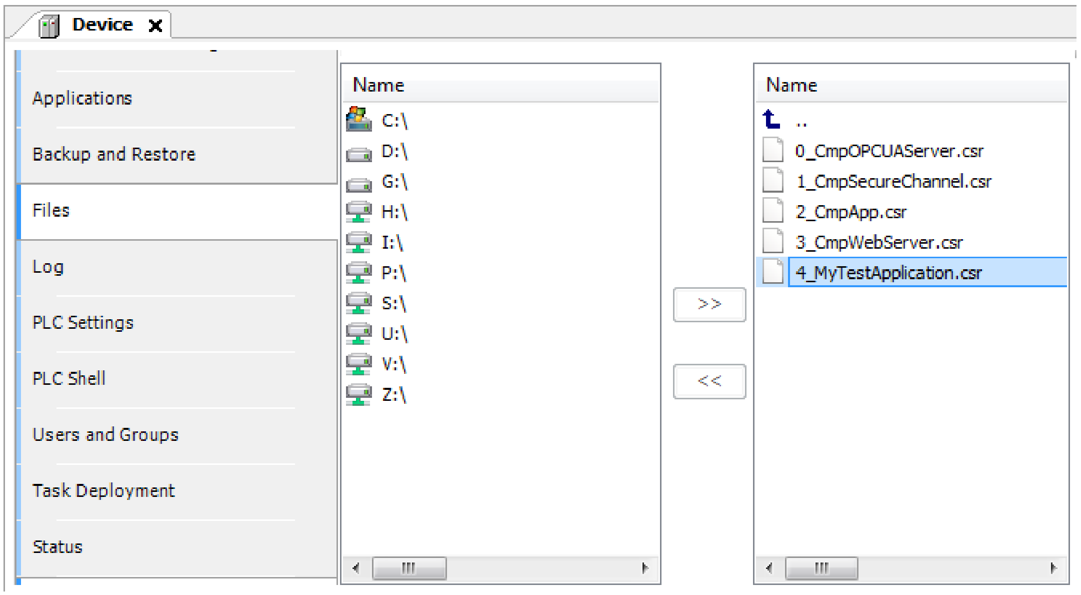
```
> cert-getcertlist own

------------------------------------------------------
List of own certificates
------------------------------------------------------
Number: 0
Thumbprint: 132245455ef12cf27ee94bd5125350bdc84fb4e5
Subjects:
 - commonName: MyTestApplication
Valid from: 25.7.2017 12:47:36
Valid until: 4.8.2017 12:47:36
```

There is always a key pair behind a certificate in the CODESYS certificate store. The secret private key and the public key associated with this certificate. In order for the public key to appear trustworthy to the communication partners, the certificate must be signed by a trusted authority. In connection with the certificate from the last section, we speak of a so called "self-signed certificate".

In order to obtain the signature of a trusted authority for this certificate, a corresponding application must be submitted. The "Certificate Signing Request" (CSR). This request can be generated as follows:

```
> cert-createcsr 4

Create CSR for application with given index. Check logger to see when finished.
```

After that the CSR can be uploaded from the PLC folder :



0_CmpOPCUAServer.csr
1_CmpSecureChannel.csr
2_CmpApp.csr
3_CmpWebServer.csr
4_MyTestApplication.csr

Certificate Access

```
IF _hComponent = RTS_INVALID_HANDLE THEN
    _hComponent := CMAddComponent(_sComponentName, _dwComponentId, _dwComponentVersion, ADR(Result)
    IF Result = Errors.ERR_OK THEN
        _hCertStore := X509CertStoreOpen(_dwComponentId, 0);
        certInfo.subject.numOfEntries := 1;
        certInfo.subject.entries := ADR(subject);
        RtsOIDCreate(ADR(KnownOIDs.RTS_OID_COMMON_NAME), ADR(subject.id));
        subject.value := ADR(_sComponentName);
        _hClaim := X509CertStoreRegister(_hCertStore, _dwComponentId, ADR(certInfo), ADR(Result));
        IF Result = Errors.ERR_OK THEN
            _hCert := X509CertStoreGetRegisteredCert(_hCertStore, _hClaim, 0);
        END_IF
    END_IF
END_IF
ProvideCertificate := (_hCert <> RTS_INVALID_HANDLE);
```

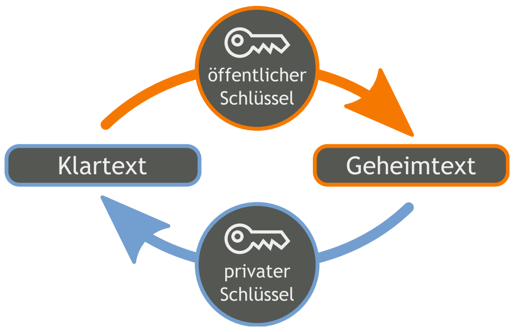
```
METHOD ReleaseCertificate
```

```
_hCert := RTS_INVALID_HANDLE;
IF _hCertStore <> RTS_INVALID_HANDLE THEN
    X509CertStoreUnregister(_hCertStore, _hClaim);
    _hClaim := RTS_INVALID_HANDLE;
    X509CertStoreClose(_hCertStore);
    _hCertStore := RTS_INVALID_HANDLE;
END_IF
IF _hComponent <> RTS_INVALID_HANDLE THEN
    CMRemoveComponent(_hComponent);
    _hComponent := RTS_INVALID_HANDLE;
END_IF
```

Encryption

```
METHOD AsymmetricEncryptMessage : ULINT
VAR_INPUT
    sPlainText : REFERENCE TO MESSAGE;
    szPlainText : ULINT;
    abyCipherText : REFERENCE TO BUFFER;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    ksPublicKey : RtsCryptoKey;
    bsPlainText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(szPlainText), pB
    bsCipherText : RtsByteString := (ui32MaxLen:=SIZEOF(BUFFER), ui32Len:=0, pByData:=ADR(abyCipher
END_VAR
```

```
Result := X509CertGetPublicKey(
    hCert:=_hCert,
    pPublicKey:=ADR(ksPublicKey)
);
Result := CryptoAsymmetricEncrypt(
    hAlgo:=_hAsymmetricCipher,
    pPlainText:=ADR(bsPlainText),
    publicKey:=ksPublicKey,
    pCipherText:=ADR(bsCipherText)
);
AsymmetricEncryptMessage := bsCipherText.ui32Len;
```



Decryption

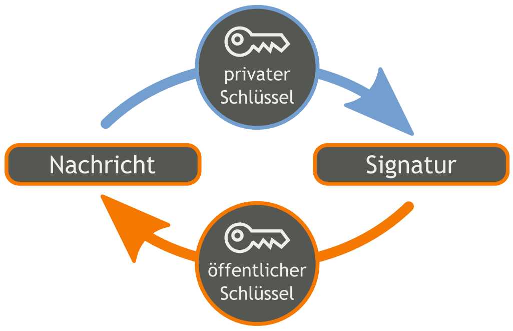
```
METHOD AsymmetricDecryptMessage : ULINT
VAR_INPUT
    abyCipherText : REFERENCE TO BUFFER;
    szCipherText : ULINT;
    sPlainText : REFERENCE TO MESSAGE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    ksPrivateKey : RtsCryptoKey;
    bsCipherText : RtsByteString := (ui32MaxLen:=SIZEOF(BUFFER), ui32Len:=TO_UDINT(szCipherText), p
    bsPlainText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=0, pByData:=ADR(sPlainTex
END_VAR
```

```
Result := X509CertGetPrivateKey(
    hCertStore:=_hCertStore,
    hCert:=_hCert,
    pPrivateKey:=ADR(ksPrivateKey)
);

Result := CryptoAsymmetricDecrypt(
    hAlgo:=_hAsymmetricCipher,
    pCipherText:=ADR(bsCipherText),
    privateKey:=ksPrivateKey,
    pPlainText:=ADR(bsPlainText)
);
AsymetricDecryptMessage := bsPlainText.ui32Len;
```

Signing

```
METHOD AsymmetricSignMessage : ULINT
VAR_INPUT
    sMessage : REFERENCE TO MESSAGE;
    szMessage : ULINT;
    abySignature : REFERENCE TO SIGNATURE;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    ksPrivateKey : RtsCryptoKey;
    bsPlainText : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(szMessage), pByD
    bsSignature : RtsByteString := (ui32MaxLen:=SIZEOF(abySignature), ui32Len:=0, pByData:=ADR(abyS
END_VAR
```

```
Result := X509CertGetPrivateKey(
    hCertStore:=_hCertStore,
    hCert:=_hCert,
    pPrivateKey:=ADR(ksPrivateKey)
);

Result := CryptoSignatureGenerate(
    hAlgo:=_hAsymmetricSigningCipher,
    pData:=ADR(bsPlainText),
    privateKey:=ksPrivateKey,
    pSignature:=ADR(bsSignature)
);
AsymmetricSignMessage := bsSignature.ui32Len;
```



Verifying

```
METHOD AsymmetricVerifyMessage : BOOL
VAR_INPUT
    sMessage : REFERENCE TO MESSAGE;
    szMessage : ULINT;
    abySignature : REFERENCE TO SIGNATURE;
    szSignature : ULINT;
END_VAR
VAR
    Result : RTS_IEC_RESULT;
    ksPublicKey : RtsCryptoKey;
    bsMessage : RtsByteString := (ui32MaxLen:=SIZEOF(MESSAGE), ui32Len:=TO_UDINT(szMessage), pByDat
    bsSignature : RtsByteString := (ui32MaxLen:=SIZEOF(SIGNATURE), ui32Len:=TO_UDINT(szSignature),
END_VAR
```

```
Result := X509CertGetPublicKey(
    hCert:=_hCert,
    pPublicKey:=ADR(ksPublicKey)
);

Result := CryptoSignatureVerify(
    hAlgo:=_hAsymmetricSigningCipher,
    pData:=ADR(bsMessage),
    publicKey:=ksPublicKey,
    pSignature:=ADR(bsSignature)
);
AsymmetricVerifyMessage := (Result = CmpErrors.Errors.ERR_OK);
```

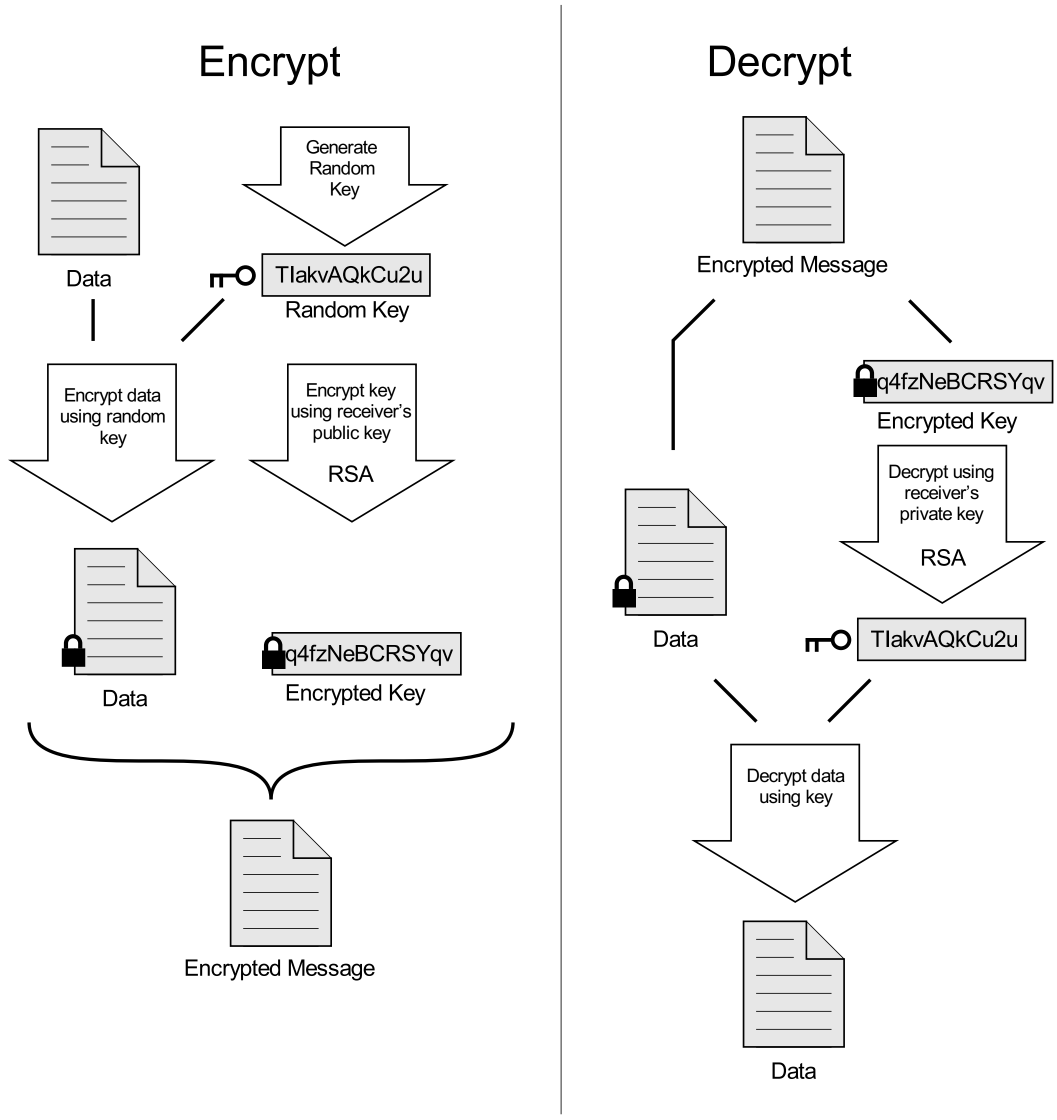
Pay by Use

In an exemplary application use case, various functionalities were implemented for a system. Some of these functions are available to all users. Other functions are only available if a certain condition is fulfilled. This special prerequisite should be able to be transferred from the outside of the system via an untrusted channel and should be active only for a certain period of time. The site that created the prerequisite for the use of the protected functions must be identifiable by the system without any doubt, only then will the protected function be activated and can be used.

A signed and encrypted document specifies which function is allowed to be used. In the further content of the document, one can use a date specification to determine how long the respective functionality can be used.

With the help of the signature, the system can clearly determine who wrote the document and whether it has undergone any changes in the meantime. Trough the (Asymmetric) Encryption only the recipient who has been determined in advance can decrypt the document and use its contents.

The following drawing is intended to illustrate the first part of the procedure:



Some notes on why which procedure was chosen:

- Using a random key for symmetric encryption of the data increases security because its value is

unpredictable.
- The use of symmetric encryption methods is recommended for larger amounts of data due to better performance.
- The use of asymmetric encryption is suitable for smaller data volumes. For this reason, only the key required for symmetric encryption of data is addressed here.
- The asymmetric signature and subsequent verification of this signature ensures that the author of the data is the intended one. As long as no one else than the sender is in control of the private key.

The following pseudo code is intended to illustrate the two steps: Creating a document and utilize a document.

```
METHOD CreateDocument : BOOL;
VAR_INPUT
    ckPublicKey : RtsCryptoKey;
    bsData : RtsByteString;
    bsDocument : REFERENCE TO RtsByteString;
END_VAR
VAR
    bsEncryptedData : RtsByteString := (ui32MaxLen:=32, ui32Len:=0, pByData:=bsDocument.pByData);
    bsEncryptedKey : RtsByteString := (ui32MaxLen:=256, ui32Len:=0, pByData:=bsDocument.pByData + 3
    bsSignature : RtsByteString := (ui32MaxLen:=256, ui32Len:=0, pByData:=bsDocument.pByData + 32 +
    bsInitVector : RtsByteString := (ui32MaxLen:=TO_UDINT(_szBlock), ui32Len:=TO_UDINT(_szBlock), p

    Result : RTS_IEC_RESULT;

    abyKey : ARRAY[0..31] OF BYTE; // 256 Bit
    bsKey : RtsByteString := (ui32MaxLen:=SIZEOF(abyKey), ui32Len:=TO_UDINT(_szKey), pbyData:=ADR(a
    ksStorage : RtsCryptoKeyStorage := (byteString:=bsKey);
    ckKey : RtsCryptoKey := (keyType:=RtsCryptoKeyType.KeyType_Key, key:=ksStorage);

    ckPrivateKey : RtsCryptoKey;
    bsCollection : RtsByteString := (ui32MaxLen:=32 + 256, ui32Len:=32 + 256, pByData:=bsDocument.p

    abyInitVector : ARRAY[0..15] OF BYTE; // Random Initial Value of Length ``_szBlock``
END_VAR
```

```
// Generate Random Key
Result := CryptoGenerateRandomNumber(ui32NumOfRandomBytes:=TO_UDINT(_szKey), pRandom:=ADR(bsKey));
// Generate Random InitVector
Result := CryptoGenerateRandomNumber(ui32NumOfRandomBytes:=TO_UDINT(_szBlock), pRandom:=ADR(bsInitV

// Symmetric Encrypt Data using Random Key
Result := CryptoSymmetricEncrypt(
    hAlgo:=_hSymmetricCryptoCipher,
    pPlainText:=ADR(bsData),
    key:=ckKey,
    pInitVector:=ADR(bsInitVector),
    xEnablePadding:=TRUE,
    pCipherText:=ADR(bsEncryptedData)
);

// Asymmetric Encrypt Random Key with Receiver's Public Key
Result := CryptoAsymmetricEncrypt(
    hAlgo:=_hAsymmetricCryptingCipher,
    pPlainText:=ADR(bsKey),
    publicKey:=ckPublicKey,
    pCipherText:=ADR(bsEncryptedKey)
);

// Asymmetric Sign the Collection of Encrypted Data and Encrypted Key with Sender's the Private Key
Result := X509CertGetPrivateKey(
    hCertStore:=_hCertStore,
    hCert:=_hCert,
    pPrivateKey:=ADR(ckPrivateKey)
);
Result := CryptoSignatureGenerate(
    hAlgo:=_hAsymmetricSigningCipher,
    pData:=ADR(bsCollection),
    privateKey:=ckPrivateKey,
    pSignature:=ADR(bsSignature)
);
```

```
METHOD UtilizeDocument
VAR_INPUT
    ckPublicKey : RtsCryptoKey;
    bsDocument : RtsByteString;
    bsData : REFERENCE TO RtsByteString;
END_VAR
VAR
    bsEncryptedData : RtsByteString := (ui32MaxLen:=32, ui32Len:=32, pByData:=bsDocument.pByData);
    bsEncryptedKey : RtsByteString := (ui32MaxLen:=256, ui32Len:=256, pByData:=bsDocument.pByData +
    bsSignature : RtsByteString := (ui32MaxLen:=256, ui32Len:=256, pByData:=bsDocument.pByData + 32
    bsInitVector : RtsByteString := (ui32MaxLen:=TO_UDINT(_szBlock), ui32Len:=TO_UDINT(_szBlock), p

    Result : RTS_IEC_RESULT;

    abyKey : ARRAY[0..31] OF BYTE; // 256 Bit
    bsKey : RtsByteString := (ui32MaxLen:=SIZEOF(abyKey), ui32Len:=0, pbyData:=ADR(abyKey));
    ksStorage : RtsCryptoKeyStorage := (byteString:=bsKey);
    ckKey : RtsCryptoKey := (keyType:=RtsCryptoKeyType.KeyType_Key, key:=ksStorage);

    ckPrivateKey : RtsCryptoKey;
    bsCollection : RtsByteString := (ui32MaxLen:=32 + 256, ui32Len:=32 + 256, pByData:=bsDocument.p


END_VAR
```

```
// Asymmetric Verify the Signature of the Collection of Encrypted Data and Encrypted Key with the S
Result := CryptoSignatureVerify(
    hAlgo:=_hAsymmetricSigningCipher,
    pData:=ADR(bsCollection),
    publicKey:=ckPublicKey,
    pSignature:=ADR(bsSignature)
);
IF Result <> ERRORS.ERR_OK THEN
    RETURN;
END_IF

Result := X509CertGetPrivateKey(
    hCertStore:=_hCertStore,
    hCert:=_hCert,
    pPrivateKey:=ADR(ckPrivateKey)
);

// Asymmetric Decrypt the Encrypted Key with the Receivers's Private Key
Result := CryptoAsymmetricDecrypt(
    hAlgo:=_hAsymmetricCryptingCipher,
    pCipherText:=ADR(bsEncryptedKey),
    privateKey:=ckPrivateKey,
    pPlainText:=ADR(bsKey)
);

// Symmetric Decrypt Data using Decrypted Key
Result := CryptoSymmetricDecrypt(
    hAlgo:=_hSymmetricCryptoCipher,
    pCipherText:=ADR(bsEncryptedData),
    key:=ckKey,
    pInitVector:=ADR(bsInitVector),
    xEnablePadding:=TRUE,
    pPlainText:=ADR(bsData)
);
```

Note

Ths methods are appropriate for handling data records in an untrusted environment. This effort is not necessary in a trusted environment like transmitting data over a TLS-Connection for the TCP-Protocol.