StreamSec ASN.1 Tools

Table of Contents

StreamSec ASN.1 Tools

1 Overview

StreamSec ASN.1 Tools has four elements:

1. An ASN.1 Design time Editor and Precompiler that integrates with the Delphi IDE and gives you Syntax Highlighting (Delphi 7 only). The precompiler outputs error messages with reference to source line and column when syntax errors are encountered.
2. A Meta Data Filer. The ASN.1 precompiler outputs data in a complex MMF format that contains the field values of the objects used for DER parsing. The Meta Data Filer is a generic technology that can be used for other purposes as a substitute for the VCL tReader and tWriter classes.
3. Generic classes for DER parsing.
4. A precompiler that generates Delphi interfaces and classes from ASN.1 and MMF files.

1.1 What is ASN.1?

ASN.1 is a language with the sole purpose of defining data formats. It is useful when you stream data from one application to another or just save data to a file.

Description

The main advantages of using ASN.1 for data definition are:

- **Overview (see page 1)** and **Documentation**. If you are writing directly to a tStream or a tWriter when saving your data the only definition of the data format will be your code. ASN.1 might help you structure your data.
- **Flexibility**. While data bases typically have an architecture that is optimized for storage of large arrays of syntactically equivalent records, ASN.1 is a better choice when dealing with heterogenous data quantities that have to be stored in a variable or complex format for compactness.
- **Interoperability**. ASN.1 is a standard for data format definitions that can be used on almost any contemporary platform.
- **Maintainability**. By separating the data format definition from your code it becomes easy to reuse data that was stored with a different application without having access to the source code for that application.
- **Extensibility** and **Compatibility**. ASN.1 contains key words such as OPTIONAL and CHOICE that can be used when adding fields to existing data formats without loosing the ability to load existing data stored on the old format.

ASN.1 is typically associated with the Basic Encoding Rules (BER) or Distinguished Encoding Rules (DER). BER and DER append a header to each field that identifies it and specifies the length of the field.

2 ASN.1 Language Reference

StreamSec ASN.1 is based on the standard 1988 ASN.1 syntax with some additions and modifications.

Basic syntax

Basic syntax	Description
The module header (see page 5)	Module header declarations in StreamSec ASN.1
Declaring a named constant (see page 6)	How to declare a named constant value
Declaring an ENUMERATED type (see page 6)	Declaring a StreamSec ASN.1 ENUMERATED type
Declaring a BIT STRING with named bits (see page 7)	How to declare a BIT STRING with named bits (Pascal set)
Declaring a constructed type (see page 7)	How to declare a simple constructed type
Declaring a tagged type (see page 8)	How to declare a tagged type
Declaring an ENCAPSULATED type (see page 9)	How to declare an ENCAPSULATED type
Declaring an OF type (see page 9)	How to declare a SEQUENCE OF or SET OF type
Declaring a primitive type with a SIZE constraint (see page 10)	How to declare a primitive type with a SIZE constraint
Declaring a CHOICE type (see page 10)	How to declare a CHOICE type
Declaring a TYPE-IDENTIFIER type (see page 11)	How to declare a TYPE-IDENTIFIER type
Extending a constructed type (see page 12)	How to extend a constructed type
Extending a CHOICE type (see page 12)	How to extend a CHOICE type
Extending a TYPE-IDENTIFIER type (see page 13)	How to extend a TYPE-IDENTIFIER type

Syntactic elements

Syntactic elements	Description
Special symbols (see page 2)	Special symbols used by StreamSec ASN.1
Primitive types (see page 3)	Primitive types supported by StreamSec ASN.1
Constructed types (see page 3)	Basic constructed types supported by StreamSec ASN.1
Variant types (see page 4)	Basic variant types supported by StreamSec ASN.1
Field keywords (see page 4)	Common keywords used in field declarations
Inheritance (see page 4)	Keywords used for constructed and variant type inheritence.
Precompiler directives (see page 5)	Precompiler directives used by StreamSec ASN.1

2.1 Syntactic elements

2.1.1 Special symbols

Special symbols used by StreamSec ASN.1

Description

Symbol	Description
::=	Definition
{ }	Begin and end compound declaration
;	End single line declaration
( )	Enclose integer constant, size range or identifier field reference
,	Separate fields in the declaration of a sequence, set, choice or enumeration
|	Separate element in the declaration of a TYPE-IDENTIFIER

[ ]	Enclose non-default DER tag
..	Used in size range expressions
. &	Used in "INSTANCE OF" field declarations
=	Assign non-default value to field
--	Comment

2.1.2 Primitive types

Primitive types supported by StreamSec ASN.1

Description

Name	Description
EOC	Never used in DER encoded data. Used in BER encoded data to mark the end of indefinite SEQUENCE OF or SET OF data.
BOOLEAN	Boolean
INTEGER	Integer of an arbitrary size.
BIT STRING	Use the BIT STRING type when the size of the data is not necessarily an integer number of octets, such as when encoding a Pascal set variable.
OCTET STRING	The generic type for raw data.
NULL	Fields of this type are always empty. NULL fields are typically used in TYPE-IDENTIFIER declarations.
OBJECT / OBJECT IDENTIFIER	OIDs are used for externally recognizable identification of formats and values. Each OID arc is assigned to a specific organization. New OIDs are defined by appending a row of integer constants to the end of the OID arc of the organization.
ENUMERATED	Typically used for encoding named integer constants, much like Pascal enumerations.
UTF8STRING	Utf8string
BMPSTRING	Big endian Unicode-16
PRINTABLESTRING	Case insensitive ASCII
UTCTIME	Two-year digit time, in the range 1950-2050
GENERALIZEDTIME	Four-year digit time.

2.1.3 Constructed types

Basic constructed types supported by StreamSec ASN.1

Description

Type	Description
SEQUENCE	A sequential record of primitive or constructed data fields
SET	A non-sequential record of primitive or constructed data fields. SET data structures must be sorted when DER encoded
SEQUENCE OF	A sequential row of records of the same primitive, constructed or variant type

SET OF	A non-sequential row of records of the same primitive, constructed or variant type. SET OF data structures must be sorted when DER encoded

2.1.4 Variant types

Basic variant types supported by StreamSec ASN.1

Description

Name	Description
ANY	Any type
CHOICE	A CHOICE type is similar to the case segment in a Pascal record. A field of a CHOICE type is encoded as the selected field in the CHOICE declaration. Each CHOICE field must have a unique DER tag
TYPE-IDENTIFIER	Similar to CHOICE types, except that the selection is identified by the value of a sibling field. TYPE-IDENTIFIER fields might have identical DER tags, but must be identified by unique external values.

2.1.5 Field keywords

Common keywords used in field declarations

Description

Keyword	Description
EXPLICIT	Used in conjunction with a DER tag to tell the DER parser that the field is encoded with both an outer and an inner tag. This directive might occur in both constructed and variant type field declarations
IMPLICIT	Used in conjunction with a DER tag to tell the DER parser that declared tag replaces the original tag of the field type. This directive might occur in both constructed and variant type field declarations
ENCAPSULATES	Equivalent to EXPLICIT but used in conjunction with the BIT STRING or OCTET STRING key words rather than []. This directive might occur in both constructed and variant type field declarations
DEFAULT	The field has a default value. The field is omitted from the DER output when the actual value is equal to the default value. This directive might occur in SEQUENCE or SET type field declarations
OPTIONAL	The field is omitted from the DER output when empty. This directive might occur in SEQUENCE or SET type field declarations
IDENTIFIED BY	This keyword is mandatory in each field of a TYPE-IDENTIFIER type declaration. The type to the left is identified by the primitive value to the right

2.1.6 Inheritance

Keywords used for constructed and variant type inheritence.

Description

Keyword	Description
EXTENDS	Used in declarations of constructed or variant types for inheriting the fields from a previously declared type

REPLACES	Optionally used in field declarations inside types that extend a previously declared type. REPLACES indicates that a field that was inherited through the EXTENDS clause is replaced by a new field declaration.

2.1.7 Precompiler directives

Precompiler directives used by StreamSec ASN.1

Description

A precompiler directive is a special kind of comment that must be inserted at the top of the *.asn module. The current version supports the following two directives:

--#STGEN MODULE ALWAYS

The associated pascal unit is generated anew each time the *.asn module is compiled.

--#STGEN MODULE IFNEW

The associated pascal unit is generated when the *.asn module is compiled only if a *.pas file with the same name does not already exist in the path.

2.2 Basic syntax

2.2.1 The module header

Module header declarations in StreamSec ASN.1

Description

Each *.asn file must contain a single module. The module has the following syntax:

[<precompiler-directives>]

<module-name> [module-oid] **DEFINITIONS** [**EXPLICIT TAGS**|**IMPLICIT TAGS**] ::=

BEGIN

[**EXPORTS** <identifier-list>;|**ALL**;]

[**IMPORTS** <identifier-list> **FROM** <module-name>;|<module-oid>;]

...

[**IMPORTS** <identifier-list> **FROM** <module-name>;|<module-oid>;]

<declarations>

END

- Any precompiler directive must occur as the first lines of the module, before the module header and any comments.
- The module "name space" is identified by either a module name or a named module oid. It is currently recommended to use only a module name that contains the same alphanumerical characters as the file name.
- The module header might specifiy that tags (e.g. [0]) are either EXPLICIT or IMPLICIT by default. The default is EXPLICIT TAGS.
- The EXPORTS clause controls which Delphi classes are generated by the precompiler.
- An IMPORTS clause must be present if the module uses declarations from another module. Note: OIDs are always imported and do not have to be listed in the IMPORTS clause.

2.2.2 Declaring a named constant

How to declare a named constant value

Description

Each constant declaration has the following syntax:

<name> <type> ::= <value-expression>

The current version of the precompiler does not convert ASN.1 constant declarations to Pascal declarations. ASN.1 constants are only used internally e.g. for setting DEFAULT values or IDENTIFIED BY values.

Example

```
one INTEGER ::= 0;
rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }
```

2.2.3 Declaring an ENUMERATED type

Declaring a StreamSec ASN.1 ENUMERATED type

Description

An ENUMERATED type is the ASN.1 equivalent to Pascal enumerations. The declaration has the following syntax:

<name> ::= **INTEGER**|**ENUMERATED** { <name1>(value1)

[, <name2>(value2)

[, <name3>(value3)

...]]};

An ENUMERATED type is converted to a Pascal enumeration.

Remarks

Enumerations with explicit value assignments are supported by Delphi 6 and up. If you are using an older version of Delphi you must make sure that the first value is zero, and that each subsequent value is one higher than the previous.

2.2.4 Declaring a BIT STRING with named bits

How to declare a BIT STRING with named bits (Pascal set)

Description

A BIT STRING with named bits is the ASN.1 equivalent to Pascal sets. The declaration has the following syntax:

<name> ::= **BIT STRING** { <name1>(value1)

[, <name2>(value2)

[, <name3>(value3)

...]]};

A BIT STRING with named bits is converted to a Pascal enumeration and a Pascal set declaration.

Example

```
KeyUsage ::= BIT STRING {
     digitalSignature        (0),
     nonRepudiation          (1),
     keyEncipherment         (2),
     dataEncipherment        (3),
     keyAgreement            (4),
     keyCertSign             (5),
     cRLSign                 (6),
     encipherOnly            (7),
     decipherOnly            (8) };
```

The generated Delphi code is:

```
  tKeyUsageEnum = (DigitalSignature,
                   NonRepudiation,
                   KeyEncipherment,
                   DataEncipherment,
                   KeyAgreement,
                   KeyCertSign,
                   CRLSign,
                   EncipherOnly,
                   DecipherOnly);
  tKeyUsage = set of tKeyUsageEnum;
```

2.2.5 Declaring a constructed type

How to declare a simple constructed type

Description

A simple constructed type is equivalent to a Pascal record. It has the following syntax:

<typename> ::= **SEQUENCE**|**SET** {

<fieldname1> <field type expression>[,

<fieldname2> <field type expression>[,

<fieldname3> <field type expression>

...]]

}

Like a type name, a field name must not contain any white characters or special symbols.

A typical field type declaration has the following syntax:

<type expression>[OPTIONAL|DEFAULT <value>|= <value>]

Fields of TYPE-IDENTIFIER types have the following syntax:

<typename>.&Type(&<identifier>)[OPTIONAL]

The typename must identify a fully defined TYPE-IDENTIFIER type. The identifier must be the name path of an INTEGER or OBJECT field, given relative to the owner. The identifier field must occur before this field in the field list.

Example

```
TBSCertificate  ::=  SEQUENCE  {
        version         [0]  EXPLICIT INTEGER { v1(0), v2(1), v3(2) } DEFAULT v1,
        serialNumber         INTEGER,
        signature            SignatureAlgorithmIdentifier,
        issuer               Name,
        validity             Validity,
        subject              Name,
        subjectPublicKeyInfo SubjectPublicKeyInfo,
        issuerUniqueID  [1]  IMPLICIT BIT STRING OPTIONAL,
                             -- If present, version shall be v2 or v3
        subjectUniqueID [2]  IMPLICIT BIT STRING OPTIONAL,
                             -- If present, version shall be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                             -- If present, version shall be v3
        }
```

A typical example of a field of a TYPE-IDENTIFIER type:

```
PKAlgorithmIdentifier ::= SEQUENCE {
     algorithm    OBJECT,
     parameters    PKAlgorithmID.&Type(&algorithm) OPTIONAL}
```

2.2.6 Declaring a tagged type

How to declare a tagged type

Description

Tagged types are typically used in order to make the DER encoding of constructed types or CHOICE types unambiguous.

The syntax for a tagged type expression is:

"["<tag value> [<cls>]"]" [**EXPLICIT**|**IMPLICIT**] <inner type expression>

The tag value must be an integer. The cls value must be either of the reserved words UNIVERSAL, CONTEXT SPECIFIC, APPLICATION or PRIVATE. The default is CONTEXT SPECIFIC. The EXPLICIT or IMPLICIT key word should only be used if different from the module default. See The module header (see page 5).

As a rule of thumb, all fields in constructed type declaration including and following the first OPTIONAL or DEFAULT field shall be tagged and have unique tags within context.

The tag must be EXPLICIT if the inner type expression identifies a CHOICE type.

Example

```
GeneralSubtree ::= SEQUENCE {
     base                    GeneralName,
     minimum         [0]     BaseDistance DEFAULT 0,
     maximum         [1]     BaseDistance OPTIONAL }
```

Tagged choice fields. Note that the tags MUST be unique within the CHOICE type:

```
DistributionPointName ::= CHOICE {
     fullName                [0]     GeneralNames,
     nameRelativeToCRLIssuer [1]     RelativeDistinguishedName }
```

2.2.7 Declaring an ENCAPSULATED type

How to declare an ENCAPSULATED type

Description

ENCAPSULATED type expression are semantically similar to EXPLICIT tag expressions in so far that the der output will get two tags instead of one. There are however two differences:

1. While EXPLICIT is always part of tag expressions (e.g. [0] EXPLICIT), the ENCAPSULATED key word always follows the OCTET STRING or BIT STRING key word
2. The outer tag of an ENCAPSULATED tag is always primitive, while the outer tag of an EXPLICIT tag type is constructed. This can be used for preventing generic DER parsers from attempting to parse the inner contents.

The syntax for an ENCAPSULATED type expression is:

<**OCTET STRING**|**BIT STRING**> **ENCAPSULATES** <inner type expression>

Example

```
Extension ::= SEQUENCE {
     extnID    OBJECT,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue    OCTET STRING ENCAPSULATES ExtnValue.&Type(&extnID)}
```

2.2.8 Declaring an OF type

How to declare a SEQUENCE OF or SET OF type

Description

A SEQUENCE OF type is equivalent to a Pascal array. A SET OF type is equivalent to a sorted list. These types have the following syntax:

<typename> ::= **SEQUENCE**|**SET** [**SIZE** (<range>)] **OF** <template type expression>

The template type expression must end with either a semi colon (;) or a } symbol when it is declared a named type. The template type expression is typically the name of a declared constructed type, the name of a declared CHOICE type or a predefined primitive type.

Example

```
Extensions ::= SEQUENCE OF Extension;
```

Using a SET OF construct in a field type expression:

```
PresentationAddress ::= SEQUENCE {
        pSelector       [0] EXPLICIT OCTET STRING OPTIONAL,
        sSelector       [1] EXPLICIT OCTET STRING OPTIONAL,
        tSelector       [2] EXPLICIT OCTET STRING OPTIONAL,
        nAddresses      [3] EXPLICIT SET SIZE (1..MAX) OF OCTET STRING }
```

Another example of a SET OF construct in a field type expression:

```
Attribute ::= SEQUENCE {
     attrType    OBJECT,
     attrValues    SET SIZE (1..MAX) OF AttrValue.&Type(&attrType)}
```

2.2.9 Declaring a primitive type with a SIZE constraint

How to declare a primitive type with a SIZE constraint

Description

String types and INTEGER types can be declared with a size constraint that limitis the minimum and maximum size of fields of that type.

<typename> ::= <universal primitive type> (SIZE (<range>));

Some primitive types always have a length that is either fixed or determined by the encoding, such as BOOLEAN, UTCTIME, GENERALIZEDTIME, ENUMERATED and REAL. It is not possible to put a SIZE constraint on these types.

Putting SIZE constraints on INTEGER fields can be used as a way of forcing the Pascal source generator to treat the field as a Delphi Integer or Int64.

Example

```
ub-pds-name-length INTEGER ::= 16
PDSName ::= PrintableString (SIZE (1..ub-pds-name-length));
```

2.2.10 Declaring a CHOICE type

How to declare a CHOICE type

Description

Next to ANY, a CHOICE type is the second most basic variant type supported by StreamSec ASN.1. The actual type of a choice value is determined by the DER tag, which means that each field in a CHOICE declaration must have a unique tag.

<typename> ::= **CHOICE** {

<fieldname1> <field type expression>[,

<fieldname2> <field type expression>[,

<fieldname3> <field type expression>

...]]

}

Example

Each predefined type has a unique DER tag. There is no need for tag expressions if each fields already is of a unique type within the CHOICE type:

```
DisplayText ::= CHOICE {
     visibleString    VisibleString  (SIZE (1..200)),
     bmpString        BMPString      (SIZE (1..200)),
     utf8String       UTF8String     (SIZE (1..200))}
```

Choice fields of types with the same tags must be tagged within the choice type:

```
DistributionPointName ::= CHOICE {
     fullName                [0]     GeneralNames,
     nameRelativeToCRLIssuer [1]     RelativeDistinguishedName }
```

2.2.11 Declaring a TYPE-IDENTIFIER type

How to declare a TYPE-IDENTIFIER type

Description

TYPE-IDENTIFIER types are slightly more complex than CHOICE types, in so far that the selected type is identified by the value of an external field.

<typename> ::= **TYPE-IDENTIFIER** {

{<field typename1> **IDENTIFIED BY** <value1>}["|"

{<field typename2> **IDENTIFIED BY** <value2>}["|"

{<field typename3> **IDENTIFIED BY** <value3>}

...]]

}

Example

```
PKAlgorithmIdentifier ::= SEQUENCE {
     algorithm    OBJECT,
     parameters    PKAlgorithmID.&Type(&algorithm) OPTIONAL}

PKAlgorithmID ::= TYPE-IDENTIFIER {
     {DomainParameters    IDENTIFIED BY dhpublicnumber}|
     {NULL            IDENTIFIED BY rsaEncryption}|
     {Dss-Params    IDENTIFIED BY id-dsa}|
     {OBJECT            IDENTIFIED BY id-ecPublicKey}|
     {Dss-Params        IDENTIFIED BY id-nr}|
```

```
        {RSAES-OAEP-params IDENTIFIED BY id-RSAES-OAEP}|
        {RSASSA-PSS-params IDENTIFIED BY id-RSASSA-PSS}|
        {RSASSA-PSS-params IDENTIFIED BY id-rw}|
        {OBJECT            IDENTIFIED BY id-ecnr}}
```

2.2.12 Extending a constructed type

How to extend a constructed type

Description

Extending constructed types is useful when there is a base class with properties or method implementations you want to reuse in other classes.

<newtypename> ::= **SEQUENCE EXTENDS** <oldtypename> {

[(<fieldname1> **REPLACES** <oldfieldname1>) <field type expression>[,

(<fieldname2> **REPLACES** <oldfieldname2>) <field type expression>[,

(<fieldname3> **REPLACES** <oldfieldname3>) <field type expression>

...]]]

[<fieldname4> <field type expression>[,

<fieldname5> <field type expression>[,

<fieldname6> <field type expression>

...]]]

}

The compiler will process the declaration as follows:

1. The meta data, including all fields in their original order, of the old type is assigned to the new type
2. If a REPLACES declaration is encountered, the old field is replaced with the new field at the original position in the field list
3. Any other field in the declaration will be appended at the end of the field list.

Example

```
Signed  ::=  SEQUENCE  {
        toBeSigned           ANY,
        signatureAlgorithm   SignatureAlgorithmIdentifier,
        signatureValue       BIT STRING  }

Certificate  ::= SEQUENCE EXTENDS Signed {
        (tbsCertificate REPLACES toBeSigned) TBSCertificate }
```

2.2.13 Extending a CHOICE type

How to extend a CHOICE type

Description

CHOICE types are extended in the same way contructed types are extended, with the exception that replaced fields cannot be renamed:

<newtypename> ::= **CHOICE EXTENDS** <oldtypename> {

[**REPLACES** <oldfieldname1> <field type expression>[,

REPLACES <oldfieldname2> <field type expression>[,

REPLACES <oldfieldname3> <field type expression>

...]]]

[<fieldname4> <field type expression>[,

<fieldname5> <field type expression>[,

<fieldname6> <field type expression>

...]]]

}

The compiler will process the declaration as follows:

1. The meta data, including all fields in their original order, of the old type is assigned to the new type
2. If a REPLACES declaration is encountered, the old field is replaced with the new field at the original position in the field list
3. Any other field in the declaration will be appended at the end of the field list.

2.2.14 Extending a TYPE-IDENTIFIER type

How to extend a TYPE-IDENTIFIER type

Description

TYPE-IDENTIFIER types are extended in the same way CHOICE types are extended, except that replaced fields are identified by identifier value instead of by field name:

<typename> ::= **TYPE-IDENTIFIER EXTENDS** <oldtypename> {

[{<field typename1> **IDENTIFIED BY REPLACES** <value1>}["|"

{<field typename2> **IDENTIFIED BY REPLACES** <value2>}["|"

{<field typename3> **IDENTIFIED BY REPLACES** <value3>}

...]]]

[{<field typename4> **IDENTIFIED BY** <value4>}["|"

{<field typename5> **IDENTIFIED BY** <value5>}["|"

{<field typename6> **IDENTIFIED BY** <value6>}

...]]]

3 Using the generated code

3.1 What is generated?

How is an ASN.1 module translated into Delphi source code?

Description

- The unit name is derived from the file name of the *.asn file
- Any module listed in an IMPORTS clause gets its corresponding Delphi unit added to the uses clause
- Primitive constants are not added to the Delphi output
- Enumerations in INTEGER, ENUMERATED and BIT STRING declarations get a corresponding Delphi enumeration
- BIT STRING types with named bits get a corresponding Delphi set type
- Constructed, CHOICE and TYPE-IDENTIFIER types get a Delphi interface and a Delphi class.
- Fields of the OBJECT type get an stDERUtils.ObjectIdentifier property
- Fields of visble string types or printable string types get a corresponding string property
- Fields of named constructed types or variable types get a corresponding interface property
- Classes corresponding to CHOICE types and TYPE-IDENTIFIER types implement each interface corresponding to any field
- An enumeration corresponding to the fields of a CHOICE type and TYPE-IDENTIFIER type is generated

Example

The following ASN.1 module from StreamSec PKI Tools...

```
--#STGEN MODULE IFNEW
PKIX-GeneralName DEFINITIONS ::=
BEGIN

EXPORTS ALL;

IMPORTS Name FROM PKIX-Name;
IMPORTS AnotherName, ORAddress, EDIPartyName FROM PKIX-OtherName;

GeneralNames ::= SEQUENCE SIZE (1..MAX) OF GeneralName;

GeneralName ::= CHOICE {
     otherName                       [0] IMPLICIT AnotherName,
     rfc822Name                      [1] IMPLICIT IA5String,
     dNSName                         [2] IMPLICIT IA5String,
     x400Address                     [3] IMPLICIT ORAddress,
     directoryName                   [4] IMPLICIT Name,
     ediPartyName                    [5] IMPLICIT EDIPartyName,
     uniformResourceIdentifier       [6] IMPLICIT IA5String,
     iPAddress                       [7] IMPLICIT OCTET STRING,
     registeredID                    [8] IMPLICIT OBJECT IDENTIFIER }

END
```

...is translated into the following Delphi code:

```
unit PkixGeneralName;

interface

uses
  SysUtils, { Borland RTL }
  stMemoryMapFiler, { StreamSec Core Tools }
```

```
    stCustomMetaData, { StreamSec Core Tools }
    stASN1, { StreamSec ASN.1 Tools }
    stDERUtils, { StreamSec Core Tools }
    PkixOtherName, { Import }
    PkixName; { Import }

  type
    iGeneralName = interface;

    tGeneralNameEnum = (gnUndefined,
                        gnOtherName,
                        gnRfc822Name,
                        gnDNsname,
                        gnX400Address,
                        gnDirectoryName,
                        gnEdiPartyName,
                        gnUniformResourceIdentifier,
                        gnIPaddress,
                        gnRegisteredId);

    tdmPkixGeneralName = class(tdmCustomMetaData)
    public
      class function ModuleRegName: string; override;
    end;

    iGeneralNames = interface(iASN1Struct)
    ['{15063232-5F68-48F5-8D34-6058EBA1E8AD}']
      function get_Items(aIndex: Integer): iGeneralName;
      function get_UniqueItems(aIndex: tGeneralNameEnum): iGeneralName;
      property Items[aIndex: Integer]: iGeneralName
          read get_Items;
      property UniqueItems[aIndex: tGeneralNameEnum]: iGeneralName
          read get_UniqueItems; default;
    end;

    tGeneralNames = class(tASN1Struct,iGeneralNames)
    protected
      function get_Items(aIndex: Integer): iGeneralName;
      function get_UniqueItems(aIndex: tGeneralNameEnum): iGeneralName;
      class function RegisterClassName: string; override;
    public
      property Items[aIndex: Integer]: iGeneralName
          read get_Items;
      property UniqueItems[aIndex: tGeneralNameEnum]: iGeneralName
          read get_UniqueItems; default;
    end;

    iGeneralName = interface(iASN1Struct)
    ['{ABF4B5D7-0E82-4AB0-B24A-C097FFBC051D}']
      function get_AsDirectoryName: iName;
      function get_AsDnsname: string;
      function get_AsEdiPartyName: iEDIPartyName;
      function get_AsIpaddress: iASN1Struct;
      function get_AsOtherName: iAnotherName;
      function get_AsRegisteredId: ObjectIdentifier;
      function get_AsRfc822Name: string;
      function get_AsUniformResourceIdentifier: string;
      function get_AsX400Address: iORAddress;
      function get_Choice: tGeneralNameEnum;
      procedure set_AsDnsname(const aValue: string);
      procedure set_AsRegisteredId(const aValue: ObjectIdentifier);
      procedure set_AsRfc822Name(const aValue: string);
      procedure set_AsUniformResourceIdentifier(const aValue: string);
      procedure set_Choice(const aValue: tGeneralNameEnum);
      property Choice: tGeneralNameEnum read get_Choice write set_Choice;
      property AsOtherName: iAnotherName read get_AsOtherName;
      property AsRfc822Name: string read get_AsRfc822Name write set_AsRfc822Name;
      property AsDnsname: string read get_AsDnsname write set_AsDnsname;
      property AsX400Address: iORAddress read get_AsX400Address;
      property AsDirectoryName: iName read get_AsDirectoryName;
      property AsEdiPartyName: iEDIPartyName read get_AsEdiPartyName;
      property AsUniformResourceIdentifier: string read get_AsUniformResourceIdentifier write
```

```
  set_AsUniformResourceIdentifier;
      property AsIpaddress: iASN1Struct read get_AsIpaddress;
      property AsRegisteredId: ObjectIdentifier read get_AsRegisteredId write
  set_AsRegisteredId;
    end;

    tGeneralName = class(tASN1Struct,
                         iGeneralName,
                         iAnotherName,
                         iORAddress,
                         iName,
                         iEDIPartyName)
    protected
      function get_AsDirectoryName: iName;
      function get_AsDnsname: string;
      function get_AsEdiPartyName: iEDIPartyName;
      function get_AsIpaddress: iASN1Struct;
      function get_AsOtherName: iAnotherName;
      function get_AsRegisteredId: ObjectIdentifier;
      function get_AsRfc822Name: string;
      function get_AsUniformResourceIdentifier: string;
      function get_AsX400Address: iORAddress;
      function get_Choice: tGeneralNameEnum;
      class function RegisterClassName: string; override;
      procedure set_AsDnsname(const aValue: string);
      procedure set_AsRegisteredId(const aValue: ObjectIdentifier);
      procedure set_AsRfc822Name(const aValue: string);
      procedure set_AsUniformResourceIdentifier(const aValue: string);
      procedure set_Choice(const aValue: tGeneralNameEnum);
    public
      property Choice: tGeneralNameEnum read get_Choice write set_Choice;
      property AsOtherName: iAnotherName read get_AsOtherName implements iAnotherName;
      property AsRfc822Name: string read get_AsRfc822Name write set_AsRfc822Name;
      property AsDnsname: string read get_AsDnsname write set_AsDnsname;
      property AsX400Address: iORAddress read get_AsX400Address implements iORAddress;
      property AsDirectoryName: iName read get_AsDirectoryName implements iName;
      property AsEdiPartyName: iEDIPartyName read get_AsEdiPartyName implements iEDIPartyName;
      property AsUniformResourceIdentifier: string read get_AsUniformResourceIdentifier write
  set_AsUniformResourceIdentifier;
      property AsIpaddress: iASN1Struct read get_AsIpaddress;
      property AsRegisteredId: ObjectIdentifier read get_AsRegisteredId write
  set_AsRegisteredId;
    end;

  implementation

  {$R *.dfm}

  { tdmPkixGeneralName }

  class function tdmPkixGeneralName.ModuleRegName: string;
  begin
    Result := 'PKIX-GeneralName';
  end;

  { tGeneralNames }

  function tGeneralNames.get_Items(aIndex: Integer): iGeneralName;
  begin
    Supports((inherited Items[aIndex]),iGeneralName,Result)
  end;

  function tGeneralNames.get_UniqueItems(aIndex: tGeneralNameEnum): iGeneralName;
  begin
    Supports(FindAddSelectedItem(Ord(aIndex)-1),iGeneralName,Result)
  end;

  class function tGeneralNames.RegisterClassName: string;
  begin
    Result := 'PKIX-GeneralName.GeneralNames';
  end;
```

```
{ tGeneralName }

function tGeneralName.get_AsDirectoryName: iName;
begin
  SelectChoice(4);
  Supports(Data,iName,Result);
end;

function tGeneralName.get_AsDnsname: string;
begin
  SelectChoice(2);
  Result := ContentAsString;
end;

function tGeneralName.get_AsEdiPartyName: iEDIPartyName;
begin
  SelectChoice(5);
  Supports(Data,iEDIPartyName,Result);
end;

function tGeneralName.get_AsIpaddress: iASN1Struct;
begin
  SelectChoice(7);
  Result := Self;
end;

function tGeneralName.get_AsOtherName: iAnotherName;
begin
  SelectChoice(0);
  Supports(Data,iAnotherName,Result);
end;

function tGeneralName.get_AsRegisteredId: ObjectIdentifier;
begin
  SelectChoice(8);
  Result := ContentAsOID;
end;

function tGeneralName.get_AsRfc822Name: string;
begin
  SelectChoice(1);
  Result := ContentAsString;
end;

function tGeneralName.get_AsUniformResourceIdentifier: string;
begin
  SelectChoice(6);
  Result := ContentAsString;
end;

function tGeneralName.get_AsX400Address: iORAddress;
begin
  SelectChoice(3);
  Supports(Data,iORAddress,Result);
end;

function tGeneralName.get_Choice: tGeneralNameEnum;
begin
  Result := tGeneralNameEnum(ChoiceIndex+1);
end;

class function tGeneralName.RegisterClassName: string;
begin
  Result := 'PKIX-GeneralName.GeneralName';
end;

procedure tGeneralName.set_AsDnsname(const aValue: string);
begin
  SelectChoice(2);
  EditContent(aValue);
end;
```

```
procedure tGeneralName.set_AsRegisteredId(const aValue: ObjectIdentifier);
begin
  SelectChoice(8);
  EditContent(aValue);
end;

procedure tGeneralName.set_AsRfc822Name(const aValue: string);
begin
  SelectChoice(1);
  EditContent(aValue);
end;

procedure tGeneralName.set_AsUniformResourceIdentifier(const aValue: string);
begin
  SelectChoice(6);
  EditContent(aValue);
end;

procedure tGeneralName.set_Choice(const aValue: tGeneralNameEnum);
begin
  if Ord(aValue) > 0 then
    SelectChoice(Ord(aValue)-1);
end;

initialization
  MetaDataList.RegisterMetaDataModule(tdmPkixGeneralName);
  tGeneralNames.RegisterAsMappedClass;
  tGeneralName.RegisterAsMappedClass;
end.
```

3.2 Loading DER data

How to load DER data into a generated class

Description

Each generated class inherits from the tASN1Struct class, which has LoadFromStream and LoadFromFile methods that can be used for loading DER encoded data.

Example

```
var
  lCert: iCertificate;
begin
  lCert := tCertificate.Create(nil);
  lCert.LoadFromFile('MyCert.cer',fmtDER);
  lCert.RenderAsText(Memo1.Lines,True,True,True,0,256);
```

Index