A Layman's Guide to a Subset of ASN.1, BER, and DER

Burton S. Kaliski Jr.
RSA Data Security, Inc.
Redwood City, CA

June 3, 1991

Abstract. *This note gives a layman's introduction to a subset of OSI's Abstract Syntax Notation One (ASN.1), Basic Encoding Rules (BER), and Distinguished Encoding Rules (DER). The particular purpose of this note is to provide background material sufficient for understanding and implementing the PKCS family of standards.*

Contents

1. Introduction 2

2. Abstract Syntax Notation One 3
- **2.1 Simple types 5**
- **2.2 Structured types 6**
- **2.3 Implicitly and explicitly tagged types 6**
- **2.4 Other types 7**

3. Basic Encoding Rules 7
- **3.1 Primitive, definite-length method 8**
- **3.2 Constructed, definite-length method 8**
- **3.3 Constructed, indefinite-length method 9**

4. Distinguished Encoding Rules 10

5. Notation and encodings for some types 10
- **5.1 Implicitly tagged types 11**
- **5.2 Explicitly tagged types 13**
- **5.3 ANY 15**
- **5.4 BIT STRING 17**
- **5.5 CHOICE 19**
- **5.6 IA5String 21**
- **5.7 INTEGER 23**

5.8 NULL 25
5.9 OCTET STRING 29
5.10 PrintableString 31
5.11 SEQUENCE 33
5.12 SEQUENCE OF 35
5.13 SET 36
5.14 SET OF 38
5.15 UTCTime 39

6. An example 41
6.1 Abstract notation 41
6.2 DER encoding 42

References 45

1. Introduction

It is a generally accepted design principle that abstraction is a key to managing software development. With abstraction, a designer can specify a part of a system without concern for how the part is actually implemented or represented. Such a practice leaves the implementation open; it simplifies the specification; and it makes it possible to state "axioms" about the part that can be proved when the part is implemented, and assumed when the part is employed in another, higher-level part. Abstraction is the hallmark of most modern software specifications.

One of the most complex systems today, and one that also involves a great deal of abstraction, is Open Systems Interconnection (OSI) [1]. OSI is an internationally standardized architecture that governs the interconnection of computers from the physical layer up to the user application layer. Objects at higher layers are defined abstractly and intended to be implemented with objects at lower layers. For instance, a service at one layer may require transfer of certain abstract objects between computers; a lower layer may provide transfer services for strings of ones and zeroes, using encoding rules to transform the abstract objects into such strings. OSI is called an open system because it supports many different implementations of the services at each layer.

OSI's method of specifying abstract objects is called Abstract Syntax Notation One (ASN.1) [2], and one set of rules for representing such objects as strings of ones and zeros is called the Basic Encoding Rules (BER) [3]. ASN.1 is a flexible notation that allows one to define a variety data types, from simple types such as integers and bit strings to structured types such as sets and sequences, as well as complex types defined in terms of others. BER describes how to represent or encode values of each ASN.1 type as a string of eight-bit octets. There is generally more than one way to BER-encode a given

value. Another set of rules, called the Distinguished Encoding Rules (DER), which is a subset of BER, gives a unique encoding to each ASN.1 value.

The purpose of this note is to describe a subset of ASN.1, BER and DER sufficient to understand and implement one OSI-based application, RSA Data Security, Inc.'s Public-Key Cryptography Standards. The features described include an overview of ASN.1, BER, and DER and an abridged list of ASN.1 types and their BER and DER encodings. Sections 2–4 give an overview of ASN.1, BER, and DER, in that order. Section 5 lists some ASN.1 types, giving their notation, specific encoding rules, examples, and comments about their application to PKCS. Section 6 concludes with an example, X.500 distinguished names.

Advanced features of ASN.1, such as macros, are not described in this note, as they are not needed to implement PKCS. For information on the other features, and for more detail generally, the reader is referred to CCITT Recommendations X.208 and X.209 [2,3], which define ASN.1 and BER.

Terminology and notation. In this note, an octet is an eight-bit unsigned integer. Bit 8 of the octet is the most significant and bit 1 is the least significant.

The following meta-syntax is used for in describing ASN.1 notation:

`BIT`	monospace denotes literal characters in the type and value notation; in examples, it denotes an octet value in hexadecimal
***n*₁**	bold italics denotes a variable
[]	bold square brackets indicate that a term is optional
{}	bold braces group related terms
|	bold vertical bar delimits alternatives with a group
…	bold ellipsis indicates repeated occurrences
=	bold equals sign expresses terms as subterms

2. Abstract Syntax Notation One

Abstract Syntax Notation One, abbreviated ASN.1, is a notation for describing abstract types and values.

In ASN.1, a type is a set of values, possibly infinite in size; a value of a given ASN.1 type is an element of that type's set of values. ASN.1 has four kinds of type: simple types, which are "atomic" and have no components; structured types, which have

components; tagged types, which are derived from other types; and other types, which include the `CHOICE` type and the `ANY` type. Types and values can be given names with the ASN.1 assignment operator (`::=`) , and those names can be used in defining other types and values.

Every ASN.1 type other than `CHOICE` and `ANY` has a tag, which consists of a class and a nonnegative tag number. ASN.1 types are abstractly the same if and only if their tag numbers are the same. In other words, the name of an ASN.1 type does not affect its abstract meaning, only the tag does. There are four classes of tag:

universal — for types whose meaning is the same in all applications; these types are only defined in X.208

application — for types whose meaning is specific to an application, such as X.500 directory services; types in two different applications may have the same application-specific tag and different meanings

private — for types whose meaning is specific to a given enterprise

context-specific — for types whose meaning is specific to a given structured type; context-specific tags are used to distinguish between component types with the same underlying tag within the context of a given structured type, and component types in two different structured types may have the same tag and different meanings

The types with universal tags are defined in X.208 [2], which also gives the types' universal tag numbers. Types with other tags are defined in many places, and are always obtained by implicit or explicit tagging (see Section 2.3). Table 1 lists some ASN.1 types and their universal-class tags.

Type	Tag number (decimal)	Tag number (hexadecimal)
`INTEGER`	2	`02`
`BIT STRING`	3	`03`
`OCTET STRING`	4	`04`
`NULL`	5	`05`
`OBJECT IDENTIFIER`	6	`06`
`SEQUENCE` and `SEQUENCE OF`	16	`10`
`SET` and `SET OF`	17	`11`
`PrintableString`	19	`13`
`IA5String`	22	`16`
`UTCTime`	23	`17`

Table 1. Some types and their universal-class tags.

ASN.1 types and values are expressed in a flexible, programming-language-like notation, with the following special rules:

- layout is not significant; multiple spaces and line breaks can be considered as a single space
- comments are delimited by pairs of hyphens (--), or a pair of hyphens and a line break
- identifiers (names of values and fields) and type references (names of types) consist of upper- and lower-case letters, digits, hyphens, and spaces; identifiers begin with lower-case letters; type references begin with upper-case letters

The following four subsections give an overview simple types, structured types, implicitly and explicitly tagged types, and other types. Section 5 describes specific types in more detail.

2.1 Simple types

Simple types are those not consisting of components; they are the "atomic" types. ASN.1 defines several; the types that are relevant to the PKCS standards are the following:

`BIT STRING` — an arbitrary string of bits (ones and zeroes)

`IA5String` — an arbitrary string of IA5 (ASCII) characters

`INTEGER` — an arbitrary integer

`NULL` — a null value

`OBJECT IDENTIFIER` — an object identifier, which is a sequence of integer components that identify an object

`OCTET STRING` — an arbitrary string of octets (eight-bit values)

`PrintableString` — an arbitrary string of printable characters

`UTCTime` — a "coordinated universal time" or Greenwich Mean Time (GMT) value

Simple types fall into two categories: string types and non-string types. `BIT STRING`, `IA5String`, `OCTET STRING`, `PrintableString`, and `UTCTime` are string types.

String types can be viewed, for the purposes of encoding, as consisting of components, where the components are substrings. This view allows one to encode a value whose

length is not known in advance (e.g., an octet string value input from a file stream) with a constructed, indefinite-length encoding (see Section 3).

The string types can be given size constraints limiting the length of values.

2.2 Structured types

Structured types are those consisting of components. ASN.1 defines four, all of which are relevant to the PKCS standards:

`SEQUENCE` — an ordered collection of one or more types

`SEQUENCE OF` — an ordered collection of zero or more occurrences of a given type

`SET` — an unordered collection of one or more types

`SET OF` — an unordered collection of zero or more occurrences of a given type

The structured types can have optional components, possibly with default values.

2.3 Implicitly and explicitly tagged types

Implicitly tagged types are those derived from other types by changing the tag of the underlying type. Implicit tagging is denoted by the ASN.1 keywords `[`***class number***`] IMPLICIT` (see Section 5.1).

Explicitly tagged types are those derived from other types by adding an outer tag to the underlying type. In effect, explicitly tagged types are structured types consisting of one component, the underlying type. Explicit tagging is denoted by the ASN.1 keywords `[`***class number***`] EXPLICIT` (see Section 5.2).

Tagging is useful to distinguish types within an application; it is also commonly used to distinguish component types within a structured type. For instance, optional components of a `SET` or `SEQUENCE` type are typically given distinct context-specific tags to avoid ambiguity.

For purposes of encoding, an implicitly tagged type is considered the same as the underlying type, except that the tag is different. An explicitly tagged type is considered like a structured type with one component, the underlying type. Implicit tags result in shorter encodings, but explicit tags may be necessary to avoid ambiguity if the tag of the underlying type is indeterminate (e.g., the underlying type is `CHOICE` or `ANY`).

2.4 Other types

Other types in ASN.1 include the `CHOICE` and `ANY` types. The `CHOICE` type denotes a union of one or more alternatives; the `ANY` type denotes an arbitrary value of an arbitrary type, where the arbitrary type is possibly defined in the registration of an object identifier or integer value.

3. Basic Encoding Rules

The Basic Encoding Rules for ASN.1, abbreviated BER, give one or more ways to represent any ASN.1 value as an octet string.

There are three methods to encode an ASN.1 value under BER, the choice of which depends on the type of value and whether the length of the value is known. The three methods are primitive, definite-length encoding; constructed, definite-length encoding; and constructed, indefinite-length encoding. Simple non-string types employ the primitive, definite-length method; structured types employ either of the constructed methods; and simple string types employ any of the methods, depending on whether the length of the value is known. Types derived by implicit tagging employ the method of the underlying type and types derived by explicit tagging employ the constructed methods.

In each method, the BER encoding has three or four parts:

Identifier octets. These identify the class and tag number of the ASN.1 value, and indicate whether the method is primitive or constructed.

Length octets. For the definite-length methods, these give the number of contents octets. For the constructed, indefinite-length method, these indicate that the length is indefinite.

Contents octets. For the primitive, definite-length method, these give a concrete representation of the value. For the constructed methods, these give the concatenation of the BER encodings of the components of the value.

End-of-contents octets. For the constructed, indefinite-length method, these denote the end of the contents. For the other methods, these are absent.

The three methods of encoding are described in the following sections.

3.1 Primitive, definite-length method

This method applies to simple types and types derived from simple types by implicit tagging. It requires that the length of the value be known in advance. The parts of the BER encoding are as follows:

Identifier octets. There are two forms: low tag number (for tag numbers between 0 and 30) and high tag number (for tag numbers 31 and greater).

Low-tag-number form. One octet. Bits 8 and 7 specify the class (see Table 2), bit 6 has value "0," indicating that the encoding is primitive, and bits 5–1 give the tag number.

Class	Bit 8	Bit 7
universal	0	0
application	0	1
context-specific	1	0
private	1	1

Table 2. Class encoding in identifier octets.

High-tag-number form. Two or more octets. First octet is as in low-tag-number form, except that bits 5–1 all have value "1." Second and following octets give the tag number, base 128, most significant digit first, with as few digits as possible, and with the bit 8 of each octet except the last set to "1."

Length octets. There are two forms: short (for lengths between 0 and 127), and long definite (for lengths between 0 and $2^{1008}-1$).

Short form. One octet. Bit 8 has value "0" and bits 7–1 give the length.

Long form. Two to 127 octets. Bit 8 of first octet has value "1" and bits 7–1 give the number of additional length octets. Second and following octets give the length, base 256, most significant digit first.

Contents octets. These give a concrete representation of the value (or the value of the underlying type, if the type is derived by implicit tagging). Details for particular types are given in Section 5.

3.2 Constructed, definite-length method

This method applies to simple string types, structured types, types derived simple string types and structured types by implicit tagging, and types derived from anything by

explicit tagging. It requires that the length of the value be known in advance. The parts of the BER encoding are as follows:

Identifier octets. As described in Section 3.1, except that bit 6 has value "1," indicating that the encoding is constructed.

Length octets. As described in Section 3.1.

Contents octets. The concatenation of the BER encodings of the components of the value:

- For simple string types and types derived from them by implicit tagging, the concatenation of the BER encodings of consecutive substrings of the value (underlying value for implicit tagging).
- For structured types and types derived from them by implicit tagging, the concatenation of the BER encodings of components of the value (underlying value for implicit tagging).
- For types derived from anything by explicit tagging, the BER encoding of the underlying value.

Details for particular types are given in Section 5.

3.3 Constructed, indefinite-length method

This method applies to simple string types, structured types, types derived simple string types and structured types by implicit tagging, and types derived from anything by explicit tagging. It does not require that the length of the value be known in advance. The parts of the BER encoding are as follows:

Identifier octets. As described in Section 3.2.

Length octets. One octet, `80`.

Contents octets. As described in Section 3.2.

End-of-contents octets. Two octets, `00 00`.

Since the end-of-contents octets appear where an ordinary BER encoding might be expected (e.g., in the contents octets of a sequence value), the 00 and 00 appear as identifier and length octets, respectively. Thus the end-of-contents octets is really the primitive, definite-length encoding of a value with universal class, tag number 0, and length 0.

4. Distinguished Encoding Rules

The Distinguished Encoding Rules for ASN.1, abbreviated DER, are a subset of BER, and give exactly one way to represent any ASN.1 value as an octet string. DER is intended for applications in which a unique octet string encoding is needed, as is the case when a digital signature is computed on an ASN.1 value. DER is defined in Section 8.7 of X.509 [4].

DER adds the following restrictions to the rules given in Section 3:

1. When the length is between 0 and 127, the short form of length must be used

2. When the length is 128 or greater, the long form of length must be used, and the length must be encoded in the minimum number of octets.

3. For simple string types and implicitly tagged types derived from simple string types, the primitive, definite-length method must be employed.

4. For structured types, implicitly tagged types derived from structured types, and explicitly tagged types derived from anything, the constructed, definite-length method must be employed.

Other restrictions are defined for particular types (such as `BIT STRING`, `SEQUENCE`, `SET`, and `SET OF`), and can be found in Section 5.

5. Notation and encodings for some types

This section gives the notation for some ASN.1 types and describes how to encode values of those types under both BER and DER.

The types described are those presented in Section 2. They are listed alphabetically here.

Each description includes ASN.1 notation, BER encoding, and DER encoding. The descriptions also explain where each type is used in PKCS and related standards. ASN.1 notation is generally only for types, although for the type `OBJECT IDENTIFIER`, value notation is given as well.

5.1 Implicitly tagged types

An implicitly tagged type is a type derived from another type by changing the tag of the underlying type.

Implicit tagging is used for optional `SEQUENCE` components with underlying type other than `ANY` throughout PKCS, and for the `extendedCertificate` alternative of PKCS #7's `ExtendedCertificateOrCertificate` type.

5.1.1 ASN.1 notation

The ASN.1 notation for implicitly tagged types is

[**[***class***]** ***number***] IMPLICIT ***Type***

class **=** UNIVERSAL | APPLICATION | PRIVATE

where ***Type*** is a type, ***class*** is an optional class name, and ***number*** is the tag number within the class, a nonnegative integer.

In ASN.1 "modules" whose default tagging method is implicit tagging, the notation [**[***class***]** ***number***] ***Type*** is also acceptable, and the keyword `IMPLICIT` is implied. For definitions stated outside a module, the explicit inclusion of the keyword `IMPLICIT` is preferable to prevent ambiguity.

If the class name is absent, then the tag is context-specific. Context-specific tags can only appear in a component of a structured or `CHOICE` type.

For example, PKCS #8's `PrivateKeyInfo` type [5] has an optional `attributes` component with an implicit, context-specific tag:

```
PrivateKeyInfo ::= SEQUENCE {
  version Version,
  privateKeyAlgorithm PrivateKeyAlgorithmIdentifier,
  privateKey PrivateKey,
  attributes [0] IMPLICIT Attributes OPTIONAL }
```

Here the underlying type is `Attributes`, the class is absent (i.e., context-specific), and the tag number within the class is 0.

5.1.2 BER encoding

The BER encoding of an implicitly tagged value can be either primitive or constructed, depending on the underlying type. The contents octets are as for the BER encoding of the underlying value.

For example, the BER encoding of the `attributes` component of a `PrivateKeyInfo` value is as follows:

- the identifier octets are `80` if the underlying `Attributes` value has a primitive BER encoding and `a0` if the underlying `Attributes` value has a constructed BER encoding
- the length and contents octets are the same as the length and contents octets of the BER encoding of the underlying `Attributes` value

5.1.3 DER encoding

The DER encoding of an implicitly tagged value can be either primitive or constructed, depending on the underlying type. The contents octets are as for the DER encoding of the underlying value.

5.2 Explicitly tagged types

Explicit tagging denotes a type derived from another type by adding an outer tag to the underlying type.

Explicit tagging is used for optional `SEQUENCE` components with underlying type `ANY` throughout PKCS, and for the `version` component of X.509's `Certificate` type [4] as revised by RFC 1114 [6].

5.2.1 ASN.1 notation

The ASN.1 notation for explicitly tagged types is

`[`**[***class***]** ***number***`] EXPLICIT` ***Type***

class **=** `UNIVERSAL` **|** `APPLICATION` **|** `PRIVATE`

where ***Type*** is a type, ***class*** is an optional class name, and ***number*** is the tag number within the class, a nonnegative integer.

If the class name is absent, then the tag is context-specific. Context-specific tags can only appear in a component of a `SEQUENCE`, `SET` or `CHOICE` type.

In ASN.1 "modules" whose default tagging method is explicit tagging, the notation [**[***class***]** ***number***] ***Type*** is also acceptable, and the keyword `EXPLICIT` is implied. For definitions stated outside a module, the explicit inclusion of the keyword `EXPLICIT` is preferable to prevent ambiguity.

For example, PKCS #7's `ContentInfo` type [7] has an optional `content` component with an explicit, context-specific tag:

```
ContentInfo ::= SEQUENCE {
  contentType ContentType,
  content [0] EXPLICIT ANY DEFINED BY contentType OPTIONAL }
```

Here the underlying type is `ANY DEFINED BY contentType`, the class is absent (i.e., context-specific), and the tag number within the class is 0.

As another example, X.509's `Certificate` type [4] (as updated by RFC 1114 [6]) has a `version` component with an explicit, context-specific tag, where the `EXPLICIT` keyword is omitted:

```
Certificate ::= ...
  version [0] Version DEFAULT v1988,
...
```

The tag is explicit because the default tagging method for the ASN.1 "module" that defines the `Certificate` type is explicit tagging.

5.2.2 BER encoding

The BER encoding of an explicitly tagged value is always constructed. The contents octets are the BER encoding of the underlying value.

For example, the BER encoding of the `content` component of a `ContentInfo` value is as follows:

- the identifier octets are `a0`
- the length octets represent the length of the BER encoding of the underlying `ANY DEFINED BY contentType` value
- the contents octets are the BER encoding of the underlying `ANY DEFINED BY contentType` value

5.2.3 DER encoding

The DER encoding of an explicitly tagged value is always constructed. The contents octets are the DER encoding of the underlying value.

5.3 ANY

The `ANY` type denotes an arbitrary value of an arbitrary type, where the arbitrary type is possibly defined in the registration of an object identifier or associated with an integer index.

The `ANY` type is used for content of a particular content type in PKCS #7's `ContentInfo` type [7], for parameters of a particular algorithm in X.509's `AlgorithmIdentifier` type [4], and for attribute values in X.501's `Attribute` and `AttributeValueAssertion` types [8]. The `Attribute` type is used by PKCS #6 [9], PKCS #7 [7], and PKCS #8 [5], and the `AttributeValueAssertion` type is used in X.501 distinguished names.

5.3.1 ASN.1 notation

The ASN.1 notation for the `ANY` type is

`ANY [DEFINED BY` ***identifier*** `]`

where ***identifier*** is an optional identifier.

In the `ANY` form, the actual type is indeterminate.

The `ANY DEFINED BY` ***identifier*** form can only appear in a component of a `SEQUENCE` or `SET` type for which ***identifier*** identifies some other component, and that other component has type `INTEGER` or `OBJECT IDENTIFIER` (or a type derived from either of those by tagging). In that form, the actual type is determined by the value of the other component, either in the registration of the object identifier value, or in a table of integer values.

For example, X.509's `AlgorithmIdentifier` type [4] has a component of type `ANY`:

```
AlgorithmIdentifier ::= SEQUENCE {
  algorithm OBJECT IDENTIFIER,
  parameter ANY DEFINED BY algorithm OPTIONAL }
```

Here the actual type of the `parameter` component depends on the value of the `algorithm` component. The actual type would be defined in the registration of object identifier values for the `algorithm` component.

5.3.2 BER encoding

The BER encoding of an `ANY` value is the BER encoding of the actual value.

For example, the BER encoding of the value of the `parameter` component is the BER encoding of the value of the actual type as defined in the registration of object identifier values for the `algorithm` component.

5.3.3 DER encoding

The DER encoding of an `ANY` value is the DER encoding of the actual value.

5.4 BIT STRING

The `BIT STRING` type denotes an arbitrary string of bits (ones and zeroes). A `BIT STRING` value can have any length, including zero. This type is a string type.

The `BIT STRING` type is used for digital signatures on extended certificates in PKCS #6's `ExtendedCertificate` type [9], for digital signatures on certificates in X.509's `Certificate` type [4], and for public keys in certificates in X.509's `SubjectPublicKeyInfo` type.

5.4.1 ASN.1 notation

The ASN.1 notation for the `BIT STRING` type is

```
BIT STRING
```

For example, X.509's `SubjectPublicKeyInfo` type [4] has a component of type `BIT STRING`:

```
SubjectPublicKeyInfo ::= SEQUENCE {
  algorithm AlgorithmIdentifier,
  publicKey BIT STRING }
```

5.4.2 BER encoding

The BER encoding of a `BIT STRING` value can be either primitive or constructed. In a primitive encoding, the first contents octet gives the number of bits by which the length of the bit string is less than the next multiple of eight (this is called the "number of unused bits"). The second and following contents octets give the value of the bit string, converted to an octet string. The conversion process is as follows:

1. The bit string is padded after the last bit with zero to seven bits of any value to make the length of the bit string a multiple of eight. If the length of the bit string is a multiple of eight already, no padding is done.

2. The padded bit string is divided into octets. The first eight bits of the padded bit string become the first octet, bit 8 to bit 1, and so on through the last eight bits of the padded bit string.

In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the bit string, where each substring except the last has a length that is a multiple of eight bits.

For example, the BER encoding of the `BIT STRING` value "011011100101110111" can be any of the following, among others, depending on the choice of padding bits, the form of length octets, and whether the encoding is primitive or constructed:

```
03 04 06 6e 5d c0                                   DER encoding

03 04 06 6e 5d e0                                   padded with "100000"

03 81 04 06 6e 5d c0                                long form of length octets

23 09                                 constructed encoding: "0110111001011101" + "11"
   03 03 00 6e 5d
   03 02 06 c0
```

5.4.3 DER encoding

The DER encoding of a `BIT STRING` value is always primitive. The contents octects are as for a primitive BER encoding, except that the bit string is padded with zero-valued bits.

For example, the DER encoding of the `BIT STRING` value "011011100101110111" is

```
03 04 06 6e 5d c0
```

5.5 CHOICE

The `CHOICE` type denotes a union of one or more alternatives.

The `CHOICE` type is used to represent the union of an extended certificate and an X.509 certificate in PKCS #7's `ExtendedCertificateOrCertificate` type [7].

5.5.1 ASN.1 notation

The ASN.1 notation for the `CHOICE` type is

`CHOICE {`
`[`***identifier*$_1$**`]` ***Type*$_1$**`,`
`...,`
`[`***identifier*$_n$**`]` ***Type*$_n$** `}`

where ***identifier*$_1$** , …, ***identifier*$_n$** are optional, distinct identifiers for the alternatives, and ***Type*$_1$**, …, ***Type*$_n$** are the types of the alternatives. The identifiers are primarily for documentation; they do not affect values of the type or their encodings in any way.

The types must have distinct tags. This requirement is typically satisfied with explicit or implicit tagging on some of the alternatives.

For example, PKCS #7's `ExtendedCertificateOrCertificate` type [7] is a `CHOICE` type:

```
ExtendedCertificateOrCertificate ::= CHOICE {
  certificate Certificate, -- X.509 certificate
  extendedCertificate [0] IMPLICIT ExtendedCertificate }
```

Here the identifiers for the alternatives are `certificate` and `extendedCertificate`, and the types of the alternatives are `Certificate` and `[0] IMPLICIT ExtendedCertificate`.

5.5.2 BER encoding

The BER encoding of a `CHOICE` value is the BER encoding of the chosen alternative. The fact that the alternatives have distinct tags makes it possible to distinguish between their BER encodings. For example, the identifier octets for the BER encoding are `30` if the chosen alternative is `certificate`, and `a0` if the chosen alternative is `extendedCertificate`.

5.5.3 DER encoding

The DER encoding of a `CHOICE` value is the DER encoding of the chosen alternative.

5.6 IA5String

The `IA5String` type denotes an arbtrary string of IA5 characters. IA5 stands for International Alphabet 5, which is the same as ASCII. An `IA5String` value can have any length, including zero. This type is a string type.

The `IA5String` type is used for electronic-mail addresses and unstructured names in PKCS #9 [10].

5.6.1 ASN.1 notation

The ASN.1 notation for the `IA5String` type is

```
IA5String
```

5.6.2 BER encoding

The BER encoding of an `IA5String` value can be either primitive or constructed. In a primitive encoding, the contents octets give the characters in the IA5 string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the IA5 string.

For example, the BER encoding of the `IA5String` value "test1@rsa.com" can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
12 0d 74 65 73 74 31 40 72 73 61 2e 63 6f 6d        DER encoding

12 81 0d                                          long form of length octets
   74 65 73 74 31 40 72 73 61 2e 63 6f 6d

32 13                            constructed encoding: "test1" + "@" + "rsa.com"
   12 05 74 65 73 74 31
   12 01 40
   12 07 72 73 61 2e 63 6f 6d
```

5.6.3 DER encoding

The DER encoding of an `IA5String` value is always primitive. The contents octets are as for a primitive BER encoding.

For example, the DER encoding of the `IA5String` value "test1@rsa.com" is

```
12 0d 74 65 73 74 31 40 72 73 61 2e 63 6f 6d
```

5.7 INTEGER

The `INTEGER` type denotes an arbitrary integer. `INTEGER` values can be positive, negative, or zero, and can have any magnitude.

The `INTEGER` type is used for version numbers throughout PKCS, cryptographic values such as modulus, exponent, and primes in PKCS #1's `RSAPublicKey` and `RSAPrivateKey` types [11] and PKCS #3's `DHParameter` type [12], a message-digest iteration count in PKCS #5's `PBEParameter` type, and version numbers and serial numbers in X.509's `Certificate` type [4].

5.7.1 ASN.1 notation

The ASN.1 notation for the `INTEGER` type is

`INTEGER` **[**`{` ***identifier*$_1$**(***value*$_1$**)** … ***identifier*$_n$**(***value*$_n$**)** `}`**]**

where ***identifier*$_1$**, …, ***identifier*$_n$** are optional distinct identifiers and ***value*$_1$**, …, ***value*$_n$** are optional integer values. The identifiers, when present, are associated with values of the type.

For example, RFC 1114's `Version` type [6] is an `INTEGER` type with identified values:

```
Version ::= INTEGER { v1988(0) }
```

The identifier `v1988` is associated with the value 0. RFC 1114's `Certificate` type uses the identifier `v1988` to give a default value of 0 for the `version` component:

```
Certificate ::= ...
  version Version DEFAULT v1988,
...
```

5.7.2 BER encoding

The BER encoding of an `INTEGER` value is always primitive. The contents octets give the value of the integer, base 256, in two's complement form, most significant digit first, with the minimum number of octets. The value 0 is encoded as a single `00` octet.

Some example BER encodings (which also happen to be DER encodings) are given in Table 3.

Integer value	BER encoding
0	`02 01 00`
127	`02 02 00 7F`
128	`02 02 00 80`
256	`02 02 01 00`
-128	`02 01 80`
-129	`02 02 FF 7F`

Table 3. Example BER encodings of `INTEGER` values.

5.7.3 DER encoding

The DER encoding of an `INTEGER` value is always primitive. The contents octets are as for a primitive BER encoding.

5.8 NULL

The `NULL` type denotes a null value.

The `NULL` type is used for algorithm parameters in several places in PKCS.

5.8.1 ASN.1 notation

The ASN.1 notation for the `NULL` type is

```
NULL
```

5.8.2 BER encoding

The BER encoding of a `NULL` value is always primitive. The contents octets are empty.

For example, the BER encoding of a `NULL` value can be either of the following, as well as others, depending on the form of the length octets:

```
05 00
```

```
05 81 00
```

5.8.3 DER encoding

The DER encoding of a `NULL` value is always primitive. The contents octets are empty.

For example, the DER encoding of a `NULL` value is always `05 00`.

5.9 OBJECT IDENTIFIER

The `OBJECT IDENTIFIER` type denotes an object identifier, which is a sequence of integer components that identify an object such as an algorithm or a directory-name attribute. An `OBJECT IDENTIFIER` value can have any number of components, and components can generally have any nonnegative value. This type is a non-string type.

`OBJECT IDENTIFIER` values are given meanings by registration authorities. Each registration authority is responsible for all sequences of components beginning with a given sequence. A registration authority typically delegates responsibility for subsets of the sequences in its domain to other registration authorities, or for particular types of object. There are always at least two components.

The `OBJECT IDENTIFIER` type is used to identify content in PKCS #7's `ContentInfo` type [7], to identify algorithms in X.509's `AlgorithmIdentifier` type, and to identify attributes in X.501's `Attribute` and `AttributeValueAssertion` types [8]. The `Attribute` type is used by PKCS #6 [9], PKCS #7 [7], and PKCS #8 [5], and the `AttributeValueAssertion` type is used in X.501 distinguished names. `OBJECT IDENTIFIER` values are defined throughout PKCS.

5.9.1 ASN.1 notation

The ASN.1 notation for the `OBJECT IDENTIFIER` type is

```
OBJECT IDENTIFIER
```

The ASN.1 notation for values of the `OBJECT IDENTIFIER` type is

{ **[***identifier***]** ***component*$_1$** … ***component*$_n$** }

component*$_i$** **=** ***identifier*$_i$** **|** ***identifier*$_i$** **(***value*$_i$** **)** **|** ***value*$_i$**

where ***identifier***, ***identifier*$_1$**, …, ***identifier*$_n$** are identifiers, and ***value*$_1$**, …, ***value*$_n$** are optional integer values.

The form without ***identifier*** is the "complete" value with all its components; the form with ***identifier*** abbreviates the beginning components with another object identifier value. The identifiers ***identifier*$_1$**, …, ***identifier*$_n$** are intended primarily for documentation, but they must correspond to the integer value when both are present. These identifiers can appear without integer values only if they are among a small set of identifiers defined in X.208 [2].

For example, the following values both refer to the object identifier assigned to RSA Data Security, Inc.:

```
{ iso(1) member-body(2) 840 113549 }
{ 1 2 840 113549 }
```

Table 4 gives some other object identifier values and their meanings.

Object identifier value	Meaning
`{ 1 2 }`	ISO member bodies
`{ 1 2 840 }`	US (ANSI)
`{ 1 2 840 113549 }`	RSA Data Security, Inc.
`{ 1 2 840 113549 1 }`	RSA Data Security, Inc. PKCS
`{ 2 5 }`	directory services (X.500)
`{ 2 5 8 }`	directory services—algorithms

Table 4. Some object identifier values and their meanings.

5.9.2 BER encoding

The BER encoding of an `OBJECT IDENTIFIER` value is always primitive. The contents octets are the concatenation of $n-1$ octet strings, where n is the number of components in the complete object identifier. Each octet string is the encoding of an integer value, base 128, most significant digit first, with as few digits as possible, and with bit 8 of each octet except the last set to "1." Let $value_1$, …, $value_n$ denote the integer values of the components in the complete object identifier. The $n-1$ "subidentifiers" from which the octet string is derived are as follows:

1. The first subidentifier is $40value_1 + value_2$. (This is reasonable since $value_1$ is limited to values 0, 1, and 2, and $value_2$ is limited to the range 0 to 39 when $value_1$ is 0 or 1, and there are always at least two components, according to X.208 [2].)

2. The ith subidentifier, $2 \le i \le n-1$, is $value_{i+1}$.

For example, the subidentifiers of RSA Data Security, Inc.'s object identifier are $42 = 40 \times 1 + 2$, 840, 113549, and 1. The encoding of 42 is `2A`, the encoding of 840 is `86 48`, the encoding of 113549 is `86 F7 0D`, and the encoding of 1 is `01`, which leads to the following BER encoding:

```
06 07 2A 86 48 86 F7 0D 01
```

5.8.3 DER encoding

The DER encoding of an `OBJECT IDENTIFIER` value is always primitive. The contents octets are as for a primitive BER encoding.

5.9 OCTET STRING

The `OCTET STRING` type denotes an arbitrary string of octets (eight-bit values). An `OCTET STRING` value can have any length, including zero. This type is a string type.

The `OCTET STRING` type is used for salt values in PKCS #5's `PBEParameter` type [13], for message digests, encrypted message digests, and encrypted content in PKCS #7 [7], and for private keys and encrypted private keys in PKCS #8 [5].

5.9.1 ASN.1 notation

The ASN.1 notation for the `OCTET STRING` type is

`OCTET STRING [SIZE ({`*size* | *size*$_1$`..`*size*$_2$`})]`

where *size*, *size*$_1$, and *size*$_2$ are optional size constraints. In the `OCTET STRING SIZE (`*size*`)` form, the octet string must have *size* octets. In the `OCTET STRING SIZE (`*size*$_1$`..`*size*$_2$`)` form, the octet string must have between size1 and size2 octets. In the `OCTET STRING` form, the octet string can have any size.

For example, PKCS #5's `PBEParameter` type [13] has a component of type `OCTET STRING`:

```
PBEParameter ::= SEQUENCE {
  salt OCTET STRING SIZE(8),
  iterationCount INTEGER }
```

Here the size of the `salt` component is always eight octets.

5.9.2 BER encoding

The BER encoding of an `OCTET STRING` value can be either primitive or constructed. In a primitive encoding, the contents octets give the value of the octet string, first octet to last octet. In a constructed encoding, the contents octets give the concatenation of the BER encodings of substrings of the `OCTET STRING` value.

For example, the BER encoding of the `OCTET STRING` value `01 23 45 67 89 ab cd ef` can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
04 08 01 23 45 67 89 ab cd ef                    DER encoding

04 81 08 01 23 45 67 89 ab cd ef          long form of length octets
```

```
24 0c                  constructed encoding: 01 23 45 67 + 89 ab cd ef
   04 04 01 23 45 67
   04 04 89 ab cd ef
```

5.9.3 DER encoding

The DER encoding of an `OCTET STRING` value is always primitive. The contents octets are as for a primitive BER encoding.

For example, the BER encoding of the `OCTET STRING` value `01 23 45 67 89 ab cd ef` is

```
04 08 01 23 45 67 89 ab cd ef
```

5.10 PrintableString

The `PrintableString` type denotes an arbitrary string of printable characters from the following character set:

```
A, B, ..., Z
a, b, ..., z
0, 1, ..., 9
(space) ' ( ) + , - . / : = ?
```

This type is a string type.

The `PrintableString` type is used in several distinguished-name attribute types in X.520 [14].

5.10.1 ASN.1 notation

The ASN.1 notation for the `PrintableString` type is

```
PrintableString
```

5.10.2 BER encoding

The BER encoding of a `PrintableString` value can be either primitive or constructed. In a primitive encoding, the contents octets give the characters in the printable string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the string.

For example, the BER encoding of the `PrintableString` value "Test User 1" can be any of the following, among others, depending on the form of length octets and whether the encoding is primitive or constructed:

```
13 0b 54 65 73 74 20 55 73 65 72 20 31              DER encoding

13 81 0b 54 65 73 74 20 55 73 65 72 20 31  long form of length octets

33 0f                                      constructed form, "Test " + "User 1"
   13 05 54 65 73 74 20
   13 06 55 73 65 72 20 31
```

5.10.3 DER encoding

The DER encoding of a `PrintableString` value is always primitive. The contents octets are as for a primitive BER encoding.

For example, the DER encoding of the PrintableString value "Test User 1" is

```
13 0b 54 65 73 74 20 55 73 65 72 20 31
```

5.11 SEQUENCE

The `SEQUENCE` type denotes an ordered collection of one or more types.

The `SEQUENCE` type is used throughout PKCS and related standards.

5.11.1 ASN.1 notation

The ASN.1 notation for the `SEQUENCE` type is

`SEQUENCE {`
[*identifier*$_1$] ***Type*$_1$** **[{**`OPTIONAL` | `DEFAULT` ***value*$_1$**}]**,**
…,
[*identifier*$_n$] ***Type*$_n$** **[{**`OPTIONAL` | `DEFAULT` ***value*$_n$**}]}**

where ***identifier*$_1$** , …, ***identifier*$_n$** are optional, distinct identifiers for the components, ***Type*$_1$**, …, ***Type*$_n$** are the types of the components, and ***value*$_1$**, …, ***value*$_n$** are optional default values for the components. The identifiers are primarily for documentation; they do not affect values of the type or their encodings in any way.

The `OPTIONAL` qualifier indicates that the value of a component is optional and need not be present in the sequence. The `DEFAULT` qualifier also indicates that the value of a component is optional, and assigns a default value to the component when the component is absent.

The types of any consecutive series of components with the `OPTIONAL` or `DEFAULT` qualifier, as well as of any component immediately following that series, must have distinct tags. This requirement is typically satisfied with explicit or implicit tagging on some of the components.

For example, the X.509's `Validity` type [4] is a `SEQUENCE` type with two components:

```
Validity ::= SEQUENCE {
  start UTCTime,
  end UTCTime }
```

Here the identifiers for the components are `start` and `end`, and the types of the components are both `UTCTime`.

5.11.2 BER encoding

The BER encoding of a `SEQUENCE` value is always constructed. The contents octets are the concatenation of the BER encodings of the values of the components of the sequence, in order of definition, with the following rules for components with the `OPTIONAL` and `DEFAULT` qualifiers:

- if the value of a component with the `OPTIONAL` or `DEFAULT` qualifier is absent from the sequence, then the encoding of that component is not included in the contents octets

- if the value of a component with the `DEFAULT` qualifier is the default value, then the encoding of that component may or may not be included in the contents octets

5.11.3 DER encoding

The DER encoding of a `SEQUENCE` value is always constructed. The contents octets are the same as the BER encoding, except that if the value of a component with the `DEFAULT` qualifier is the default value, the encoding of that component is not included in the contents octets.

5.12 SEQUENCE OF

The `SEQUENCE OF` type denotes an ordered collection of zero or more occurrences of a given type.

The `SEQUENCE OF` type is used in X.501 distinguished names [8].

5.12.1 ASN.1 notation

The ASN.1 notation for the `SEQUENCE OF` type is

`SEQUENCE OF` ***Type***

where ***Type*** is a type.

For example, X.501's `RDNSequence` type [8] consists of zero or more occurences of the `RelativeDistinguishedName` type, most significant occurrence first:

```
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName
```

5.12.2 BER encoding

The BER encoding of a `SEQUENCE OF` value is always constructed. The contents octets are the concatenation of the BER encodings of the values of the occurrences in the collection, in order of occurence.

5.12.3 DER encoding

The DER encoding of a `SEQUENCE OF` value is always constructed. The contents octets are the concatenation of the DER encodings of the values of the occurrences in the collection, in order of occurence.

5.13 SET

The `SET` type denotes an unordered collection of one or more types.

The `SET` type is not used in PKCS.

5.13.1 ASN.1 notation

The ASN.1 notation for the `SET` type is

`SET {`
  [*identifier*$_1$] ***Type*$_1$** **[{**`OPTIONAL` **|** `DEFAULT` ***value*$_1$**}]**,
  …,
  [*identifier*$_n$] ***Type*$_n$** **[{**`OPTIONAL` **|** `DEFAULT` ***value*$_n$**}]**`}`

where ***identifier*$_1$** , …, ***identifier*$_n$** are optional, distinct identifiers for the components, ***Type*$_1$**, …, ***Type*$_n$** are the types of the components, and ***value*$_1$**, …, ***value*$_n$** are optional default values for the components. The identifiers are primarily for documentation; they do not affect values of the type or their encodings in any way.

The `OPTIONAL` qualifier indicates that the value of a component is optional and need not be present in the set. The `DEFAULT` qualifier also indicates that the value of a component is optional, and assigns a default value to the component when the component is absent.

The types must have distinct tags. This requirement is typically satisfied with explicit or implicit tagging on some of the components.

5.13.2 BER encoding

The BER encoding of a `SET` value is always constructed. The contents octets are the concatenation of the BER encodings of the values of the components of the set, in any order, with the following rules for components with the `OPTIONAL` and `DEFAULT` qualifiers:

- if the value of a component with the `OPTIONAL` or `DEFAULT` qualifier is absent from the set, then the encoding of that component is not included in the contents octets
- if the value of a component with the `DEFAULT` qualifier is the default value, then the encoding of that component may or may not be included in the contents octets

5.13.3 DER encoding

The DER encoding of a `SET` value is always constructed. The contents octets are the same as for the BER encoding, except that:

1. If the value of a component with the `DEFAULT` qualifier is the default value, the encoding of that component is not included.

2. There is an order to the components, namely ascending order by tag.

5.14 SET OF

The `SET OF` type denotes an unordered collection of zero or more occurrences of a given type.

The `SET OF` type is used for sets of attributes in PKCS #6 [9], PKCS #7 [7], and PKCS #8 [5], for sets of message-digest algorithm identifiers, signer information, and recipient information in PKCS #7 [7], and in X.501 distinguished names [8].

5.14.1 ASN.1 notation

The ASN.1 notation for the `SET OF` type is

`SET OF` ***Type***

where ***Type*** is a type.

For example, X.501's `RelativeDistinguishedName` type [8] consists of zero or more occurrences of the `AttributeValueAssertion` type, where the order is unimportant:

```
RelativeDistinguishedName ::= SET OF AttributeValueAssertion
```

5.14.2 BER encoding

The BER encoding of a `SET OF` value is always constructed. The contents octets are the concatenation of the BER encodings of the values of the occurrences in the collection, in any order.

5.14.3 DER encoding

The DER encoding of a `SET OF` value is always constructed. The contents octets are the same as for the BER encoding, except that there is an order, namely ascending lexicographic order of BER encoding. Lexicographic comparison of two different BER encodings is done as follows: Logically pad the shorter BER encoding after the last octet with dummy octets that are smaller in value than any normal octet. Scan the BER encodings from left to right until a difference is found. The smaller-valued BER encoding is the one with the smaller-valued octet at the point of difference.

5.15 UTCTime

The `UTCTime` type denotes a "coordinated universal time" or Greenwich Mean Time (GMT) value. A `UTCTime` value includes the local time precise to either minutes or seconds, and an offset from GMT in hours and minutes. It takes any of the following forms:

YYMMDDhhmm`Z`
YYMMDDhhmm+hh'mm'
YYMMDDhhmm−hh'mm'
YYMMDDhhmmss`Z`
YYMMDDhhmmss+hh'mm'
YYMMDDhhmmss−hh'mm'

where:

YY is the least significant two digits of the year

MM is the month (01 to 12)

DD is the day (01 to 31)

hh is the hour (00 to 23)

mm are the minutes (00 to 59)

ss are the seconds (00 to 59)

`Z` indicates that local time is GMT, + indicates that local time is later than GMT, and − indicates that local time is earlier than GMT

hh' is the absolute value of the offset from GMT in hours

mm' is the absolute value of the offset from GMT in minutes

This type is a string type.

The `UTCTime` type is used for signing times in PKCS #9's signing-time attribute [10] and for certificate validity periods in X.509's `Validity` type [4].

5.15.1 ASN.1 notation

The ASN.1 notation for the `UTCTime` type is

```
UTCTime
```

5.15.2 BER encoding

The BER encoding of a `UTCTime` value can be either primitive or constructed. In a primitive encoding, the contents octets give the characters in the string, encoded in ASCII. In a constructed encoding, the contents octets give the concatenation of the BER encodings of consecutive substrings of the string. (The constructed encoding is not particularly interesting, since `UTCTime` values are so short, but the constructed encoding is permitted.)

For example, the time this sentence was written was 4:45:40 p.m. Pacific Daylight Time on May 6, 1991, which can be represented with either of the following `UTCTime` values, among others:

"910506164540-0700"

"910506234540Z"

These values have the following BER encodings, among others:

```
17 0d 39 31 30 35 30 36 32 33 34 35 34 30 5A

17 11 39 31 30 35 30 36 31 36 34 35 34 30 2D 30 37 30 30
```

5.15.3 DER encoding

The DER encoding of a `UTCTime` value is always primitive. The contents octets are as for a primitive BER encoding.

6. An example

This section gives an example of ASN.1 notation and DER encoding: the X.501 type `Name` [8].

6.1 Abstract notation

This section gives the ASN.1 notation for the X.501 type `Name`.

```
Name ::= CHOICE {
  RDNSequence }

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET OF AttributeValueAssertion

AttributeValueAssertion ::= SEQUENCE {
   AttributeType,
   AttributeValue }

AttributeType ::= OBJECT IDENTIFIER

AttributeValue ::= ANY
```

The `Name` type identifies an object in an X.500 directory [15]. `Name` is a `CHOICE` type consisting of one alternative: `RDNSequence`.

The `RDNSequence` type gives a path through an X.500 directory tree starting at the root. `RDNSequence` is a `SEQUENCE OF` type consisting of zero or more occurences of `RelativeDistinguishedName`.

The `RelativeDistinguishedName` type gives a unique name to an object relative to the object superior to it in the directory tree. `RelativeDistinguishedName` is a `SET OF` type consisting of zero or more occurrences of `AttributeValueAssertion`.

The `AttributeValueAssertion` type assigns a value to some attribute of a relative distinguished name, such as country name or common name. `AttributeValueAssertion` is a `SEQUENCE` type consisting of two components, an `AttributeType` type and an `AttributeValue` type.

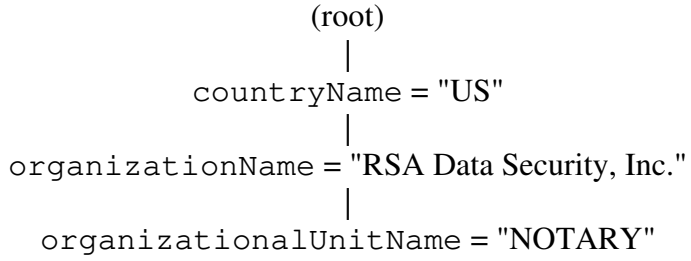
The `AttributeType` type identifies an attribute by object identifier.

The `AttributeValue` type gives an arbitrary attribute value. (The actual type of the attribute value is intended to be determined by the attribute type, even though `ANY DEFINED BY` is not employed.)

6.2 DER encoding

This section gives an example of a DER encoding of a value of type `Name`, working from the bottom up.

The name is that of RSA Data Security, Inc.'s "NOTARY" unit for Internet Privacy-Enhanced Mail [16,17], which is represented by the following path:



Each level corresponds to one `RelativeDistinguishedName` value, each of which happens for this name to consist of one `AttributeValueAssertion` value. The `AttributeType` value is before the equals sign, and the `AttributeValue` value (a printable string for the given attribute types) is after the equals sign.

6.2.1 AttributeType

The DER encodings of the three `AttributeType` values follow the primitive, definite-length method, resulting in the following octet strings:

```
06 03 55 04 06                                        countryName

06 03 55 04 0a                                    organizationName

06 03 55 04 0b                                organizationalUnitName
```

Here `countryName`, `organizationName`, and `organizationalUnitName` are X.520 attribute types [14], which have the following definitions:

```
attributeType OBJECT IDENTIFIER ::=
  { joint-iso-ccitt(2) ds(5) 4 }

countryName OBJECT IDENTIFIER ::= { attributeType 6 }
organizationName OBJECT IDENTIFIER ::= { attributeType 10 }
```

```
organizationalUnitName OBJECT IDENTIFIER ::=
  { attributeType 11 }
```

The identifier octets follow the low-tag form, since the tag is 6 for `OBJECT IDENTIFIER`. Bits 8 and 7 have value "0," indicating universal class, and bit 6 has value "0," indicating that the encoding is primitive. The length octets follow the short form. The contents octets are the concatenation of three octet strings derived from subidentifiers: $85 = 40 \times 2 + 5$; 4; and 6, 10, or 11.

6.2.2 AttributeValue

The DER encodings of the three `AttributeValue` values follow the primitive, definite-length method, resulting in the following octet strings:

```
13 02 55 53                                                    "US"

13 17 52 53 41 20 44 61 74 61                  "RSA Data Security, Inc."
      20 53 65 63 75 72 69 74
      79 2c 20 49 6e 63 2e

13 06 4e 4f 54 41 52 59                                        "NOTARY"
```

The identifier octets follow the low-octet form, since the tag is 19 for `PrintableString`. Bits 8 and 7 have value "0," indicating universal class, and bit 6 has value "0," indicating that the encoding is primitive. The length octets follow the short form, and the contents octets are the ASCII representation of the attribute value.

6.2.3 AttributeValueAssertion

The DER encodings of the three `AttributeValueAssertion` values follow the constructed, definite-length method, resulting in the following abbreviated octet strings:

```
30 09                                          countryName = "US"
   06 03 55 04 06
   13 02 55 53

30 1e                    organizationName = "RSA Data Security, Inc."
   06 03 55 04 0a
   13 17 ... 63 2e

30 0d                       organizationalUnitName = "NOTARY"
   06 03 55 04 0b
   13 06 ... 52 59
```

The identifier octets follow the low-octet form, since the tag is 16 for `SEQUENCE`. Bits 8 and 7 have value "0," indicating universal class, and bit 6 has value "1," indicating that the encoding is constructed. The length octets follow the short form, and the contents octets are the concatenation of the DER encodings of the `attributeType` and `attributeValue` components.

6.2.4 RelativeDistinguishedName

The DER encodings of the three `RelativeDistinguishedName` values follow the constructed, definite-length method, and result in the following abbreviated octet strings:

```
31 0b
   30 09 ... 55 53

31 20
   30 1e ... 63 2e

31 0f
   30 0d ... 52 59
```

The identifier octets follow the low-octet form, since the tag is 17 for `SET OF`. Bits 8 and 7 have value "0," indicating universal class, and bit 6 has value "1," indicating that the encoding is constructed. The lengths octets follow the short form, and the contents octets are the DER encodings of the respective `AttributeValueAssertion` values, since there is only one value in each set.

6.2.5 RDNSequence

The DER encoding of the `RDNSequence` value follows the constructed, definite-length method, resulting in the following abbreviated octet string:

```
30 40
   31 0b ... 55 53
   31 20 ... 63 2e
   31 0f ... 52 59
```

The identifier octets follow the low-octet form, since the tag is 16 for `SEQUENCE OF`. Bits 8 and 7 have value "0," indicating universal class, and bit 6 has value "1," indicating that the encoding is constructed. The lengths octets follow the short form, and the contents octets are the concatenation of the DER encodings of the three `RelativeDistinguishedName` values, in order of occurrence.

6.2.6 Name

The DER encoding of the `Name` value is the same as the DER encoding of the `RDNSequence` value. The final encoding is the following:

```
30 40
   31 0b
      30 09
         06 03 55 04 06
         13 02 55 53
   31 20
      30 1e
         06 03 55 04 0a
         13 17 52 53 41 20 44 61 74 61 20 53 65 63 75 72 69
               74 79 2c 20 49 6e 63 2e
   31 0f
      30 0d
         06 03 55 04 0b
         13 06 4e 4f 54 41 52 59
```

References

[1] CCITT. *Recommendation X.200: Reference Model of Open Systems Interconnection for CCITT Applications.* 1984.

[2] CCITT. *Recommendation X.208: Specification of Abstract Syntax Notation One (ASN.1).* 1988.

[3] CCITT. *Recommendation X.209: Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).* 1988.

[4] CCITT. *Recommendation X.509: The Directory—Authentication Framework.* 1988.

[5] RSA Data Security, Inc. *PKCS #8: Private-Key Information Syntax Standard.* Version 1.1, June 1991.

[6] S. Kent and J. Linn. *RFC 1114: Privacy Enhancement for Internet Electronic Mail: Part II -- Certificate-Based Key Management.* August 1989. See also [16].

[7] RSA Data Security, Inc. *PKCS #7: Cryptographic Message Syntax Standard.* Version 1.4, June 1991.

[8] CCITT. *Recommendation X.501: The Directory—Models.* 1988.

[9] RSA Data Security, Inc. *PKCS #6: Extended-Certificate Syntax Standard.* Version 1.4, June 1991.

[10] RSA Data Security, Inc. *PKCS #9: Selected Attribute Types.* Version 1.0, June 1991.

[11] RSA Data Security, Inc. *PKCS #1: RSA Encryption Standard.* Version 1.4, June 1991.

[12] RSA Data Security, Inc. *PKCS #3: Diffie-Hellman Key-Agreement Standard.* Version 1.3, June 1991.

[13] RSA Data Security, Inc. *PKCS #5: Password-Based Encryption Standard.* Version 1.4, June 1991.

[14] CCITT. *Recommendation X.520: The Directory—Selected Attribute Types.* 1988.

[15] CCITT. *Recommendation X.500: The Directory—Overview of Concepts, Models and Services.* 1988.

[16] S. Kent. *RFC [1114B]: Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management.* Draft, February 1991.

[17] B. Kaliski. *RFC [FORMS-B]: Privacy Enhancement for Internet Electronic Mail: Part IV -- Notary, Co-Issuer, CRL-Storing, and CRL-Retrieving Services.* Draft, May 1991.