



Mapping an ASN.1 Schema to a Google Protocol Buffers Schema

Objective Systems, Inc.

September 29, 2026

Copyright Notice

Copyright ©2026 Objective Systems, Inc. All rights reserved.

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information

Comments, suggestions, and inquiries regarding this document may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

1	Introduction to the Mapping.....	1
1.1	Protobuf 3 Syntax.....	1
1.2	Indecipherable Data.....	1
1.3	Version Compatibility.....	1
1.4	Name Mutation.....	1
1.5	Circular Module Imports.....	2
1.6	User Inputs.....	2
2	Definitions.....	3
3	General Procedures.....	4
3.1	Converting ASN.1 name to TitleCase.....	4
3.2	Converting ASN.1 name to UPPER_SNAKE_CASE.....	4
3.3	Converting ASN.1 name to enum Prefix.....	5
3.4	Converting ASN.1 name to lower_snake_case.....	5
3.5	Making Snake Names Unique.....	6
4	asn1_static.proto.....	7
5	Protobuf Schema Generation.....	8
6	Module Mapping.....	9
7	Types.....	10
7.1	Type Assignments.....	10
7.1.1	Mapping a Named Type.....	10
7.1.2	Procedure to Generate Wrapper Message for Type Assignment.....	13
7.1.3	Procedure to Generate Enum for Type Assignment.....	13
7.2	BOOLEAN.....	14
7.3	INTEGER.....	14
7.4	ENUMERATED.....	15
7.4.1	Procedure to Generate enum Fields.....	15
7.5	BIT STRING.....	16
7.6	OCTET STRING.....	17
7.7	NULL.....	17
7.8	DefinedType.....	17
7.9	SEQUENCE.....	17
7.9.1	Procedure to Generate Message Content from SEQUENCE.....	17
7.9.2	DEFAULT values.....	20
7.9.3	Example.....	20
7.10	SEQUENCE OF T.....	22
7.10.1	Procedure to Generate Message Content for SEQUENCE OF T.....	22
7.11	SET and SET OF.....	23
7.12	CHOICE.....	23
7.12.1	Procedure to Generate Message Content for CHOICE.....	23
7.12.2	Encoding an Extension Addition.....	24
7.12.3	Example.....	24
7.13	OBJECT IDENTIFIER.....	25
7.14	UNSUPPORTED TYPES.....	25
7.15	Character Strings.....	25

7.16 Open Type.....	26
7.16.1 Procedure to Generate Message Content From Open Type.....	26
7.17 Parameterized Types.....	28

1 Introduction to the Mapping

Objective System's ASN.1 compiler (asn1c) can generate a Google Protocol Buffers (protobuf) schema, and protobuf encoding functions, from an ASN.1 schema. This document describes how asn1c maps an ASN.1 schema to protobuf.

1.1 Protobuf 3 Syntax

The mapping presented here produces a schema using protobuf 3 syntax. However, it aims to maximize future compatibility with protobuf editions syntax in two ways: 1) by using name formats based on protobuf editions requirements and 2) by using optional on fields that are not repeated.

1.2 Indecipherable Data

In those cases where an ASN.1 decoder might encounter blocks of data that it cannot decode, the protobuf schema does not provide a way to carry the unknown ASN.1 data; such data is thus forced to be discarded when converting from an ASN.1 encoding to a protobuf encoding. This applies to:

1. unknown extension additions in a CHOICE type
2. unknown extension additions in a SEQUENCE type
3. unknown actual type for an open type

1.3 Version Compatibility

The protobuf schemas generated from two different versions of the same ASN.1 schema will, in general, NOT have compatible encodings. In particular, compatibility breaks when:

1. additions are made to an extensible CHOICE
2. additions are made to an extensible object set

1.4 Name Mutation

Though arguably unlikely, it is possible that the protobuf schema generated from one version of an ASN.1 schema will use different names for the same ASN.1 entity than the schema generated from a different version of the ASN.1. Fundamentally, this is a consequence of converting ASN.1 names to protobuf names (which follow a different format): the conversion procedure can produce name conflicts that were not in the ASN.1; it must subsequently resolve those conflicts. When such conflicts do occur, rearranging the ASN.1 schema text can change the resolution and thus the final, generated name. Likewise, editing an ASN.1 schema could introduce new conflicts, thus altering the final, generated name. Fortunately, altering names does not alter the protobuf binary encoding, but it will alter code generated from the protobuf schema.

Conflict example: "Abc-Xyz" and "AbcXyz" would both naturally map to "AbcXyz" in TitleCase format, so one of them must use a different name.

1.5 Circular Module Imports

Circular module imports are not supported. That is, the mapping does not handle the case where an ASN.1 module X imports from another module Y which, directly or indirectly, itself imports from module X. (Note: this does not prevent types from having a circular definition.) This limitation is due to a limitation in protobuf (imports in a .proto file cannot be circular).

1.6 User Inputs

User inputs play a role in the mapping. This happens in two places:

1. Designation of types as "PDU" triggers generation of an associated message type in those cases where the fundamental representation is not message.
2. Configuration of C integer representation for INTEGER types affects the choice of the protobuf type.

2 Definitions

associated protobuf message type (or "protobuf message type", for short): The associated protobuf message type for an ASN.1 type, if there is one, is simply either the fundamental type for that ASN.1 type (where the fundamental type is a message type) or a wrapper around the fundamental type (where the fundamental type is a scalar or enum). Multiple ASN.1 types may use the same protobuf message type.

base type: The base type for an ASN.1 type is the type's ultimate type. The base type for a type defined by a type assignment is the base type of the type on the right hand side of the type assignment. The base type for a (possibly constrained) built-in type (INTEGER, SEQUENCE, etc.) is the type itself. For example, given:

```
NameX ::= FirstName
FirstName ::= BIT STRING (SIZE(0..10))
```

the base type for the constrained BIT STRING is the constrained BIT STRING itself, which is also the base type for types NameX and FirstName.

fundamental representation: Every ASN.1 type has a fundamental representation in Protobuf, which is scalar, enum, message, or none. Naturally, those ASN.1 types whose fundamental representation is message will have an associated message type (namely, the fundamental type), but those ASN.1 types whose fundamental representation is scalar or enum can also have an associated message type (one which simply wraps the associated scalar or enum type).

fundamental protobuf type: The fundamental protobuf type for an ASN.1 type is the particular scalar, enum, or message used to represent the ASN.1 type when the ASN.1 type's fundamental representation is scalar, enum, or message, respectively. The fundamental type is either **generated** or **static**; it is static if and only if the fundamental representation is scalar or the representation is message and the message type is defined in package `asn1_static`.

scalar: Defined as for protobuf. It is one of the built-in types (not message or enum).

3 General Procedures

3.1 Converting ASN.1 name to TitleCase

This procedure converts an ASN.1 identifier (e.g. a component name) or ASN.1 typereference (a type name) to a unique TitleCase name, the format used for message and enum names. Optionally, the name can be made unique within a given scope, which must be identified when this procedure is invoked.

```
tcase = ASN.1 name
wherever <uppercase> <hyphen> <uppercase> occurs
    consider the longest sequence of uppercase letters before the hyphen
    let n be the length of that sequence
    if n = 1, change the one character in the sequence to lowercase
    else change all but the first character in the sequence to lowercase
wherever <hyphen> <lowercase> occurs
    covert the lowercase letter to uppercase
replace hyphen in any <digit><hyphen><digit> with an 'x'.
remove hyphens
covert first letter to uppercase
if tcase must be unique in a given scope
    let names = set of names already converted in the given scope
    if tcase is not in names, use tcase
    else
        if tcase ends in digit append 'x'
        let x = smallest integer >= 2 such that tcase + x is not in names
        append x to tcase
```

Examples:

```
userPlaneFailureType → UserPlaneFailureType
nG-DL-UP-TNL-Information → NgDlUpTnlInformation
continueROHC → ContinueROHC
gTPTunnel → GTPTunnel
ROHC-Profiles → ROhcProfiles
e-RAB-MaximumBitrateDL → ERabMaximumBitrateDL
data-forwarding-tunnel-information → DataForwardingTunnelInformation
UE-DTX-Cycle1-2ms → UeDtxCycle1x2ms
ab-X → AbX
ab-x → AbX2 (with uniqueness rule applied)
cycle1-2 → Cycle1x2 (not Cycle12)
```

3.2 Converting ASN.1 name to UPPER_SNAKE_CASE

This procedure converts an ASN.1 identifier (e.g. a component name) or typereference (a type name) to a name in UPPER_SNAKE_CASE format.

```
usnake = ASN.1 name
wherever a lowercase letter is followed by uppercase letter
    insert _ between lowercase and uppercase letter
convert all lowercase to uppercase
replace any '0-0' with '0X0'
remove hyphen in any <letter><hyphen><digit>
```

replace hyphen in any <digit><hyphen><digit> with an 'X'.
replace all hyphens with underscores

Examples:

SaneValue → SANE_VALUE
Uno-0 → UNOX0
Thing-123-Item → THING123_ITEM
B4_5 → B4X5 (not B4_5 or B45)

3.3 Converting ASN.1 name to enum Prefix

This procedure converts an ASN.1 identifier (e.g. a component name) or typereference (a type name) to an enum prefix in UPPER_SNAKE_CASE format, the format used for enum field names. The prefix will need to be unique in a given scope.

For the conversion to UPPER_SNAKE_CASE, refer to *3.2 Converting ASN.1 name to UPPER_SNAKE_CASE*.

For making the name unique, refer to *3.5 Making Snake Names Unique*.

usnake = ASN.1 typereference converted to UPPER_SNAKE_CASE
replace any _Z_ with just Z
usnake = usnake made unique in given scope
append _Z_ to usnake

Note: the sequence "_Z_" is used to separate the enum prefix from the rest of the enum field name.

3.4 Converting ASN.1 name to lower_snake_case

This procedure converts an ASN.1 identifier (e.g. a component name) or typereference (e.g. a module name) to a name in lower_snake_case format.

snake = ASN.1 name
wherever an uppercase letter is preceded by a lowercase letter
 if the lowercase letter is the first in the identifier, do nothing
 else insert underscore between them
convert all uppercase to lowercase
replace any 'l-1' with 'lx1'
remove hyphen in any <letter><hyphen><digit>
replace hyphen in any <digit><hyphen><digit> with an 'x'.
replace all hyphens with underscores

Examples:

userPlaneFailureType → user_plane_failure_type
ng-DL-UP-TNL-Information → ng_dl_up_tnl_information
continueROHC → continue_rohc
gTPTunnel → gtptunnel
rOHC-Profiles → roch_profiles
c-128-NEA1 → c128_nea1
ab-X → ab_x
ab-x → ab_x2 (with uniqueness rule applied)
All-1 → allx1 (not all1)

B4-5 → b4x5 (not b4_5 or b45)

3.5 Making Snake Names Unique

This procedure is used to make an LOWER/UPPER_SNAKE_CASE name unique in a given scope.

```
let snake = name to make unique
let names = set of names already assigned in given scope
if snake is not in names, use snake
else
  if snake ends in digit
    append 'X' (for upper names) or 'x' (for lower names)
  let x = smallest integer >= 2 such that snake + x is not in names
  append x to snake
```

Examples (with 2 appended):

MY_NAME34 → MY_NAME34X2 (not MY_NAME34_2 or MY_NAME342)

MY_NAME → MY_NAME2

my_name → my_name2

4 **asn1_static.proto**

There will be one static protobuf file, named `asn1_static.proto`. It provides definitions that are not dependent upon ASN.1 definitions. The contents are as follows:

```
syntax = "proto3";
package = asn1_static;

message Asn1BitString {
  /* number of unused bits in the final byte */
  optional uint32 unused_bits = 1;
  optional bytes data = 2;
}

message Asn1ObjectIdentifier {
  repeated uint64 arcs = 1;
}
```

Every `.proto` file that uses one of the above types **MUST** import this `.proto` file.

5 Protobuf Schema Generation

A single .proto file is generated for each ASN.1 module in the ASN.1 schema. This section describes the procedure for doing this.

The ASN.1 modules are ordered, first by module name and then by the module's defined OID (to break ties if two modules had the same name).

The generation steps are as follows:

1. Add "asn1_static" to the list of names already used as a package name.
2. For each module, in order, assign a Protobuf package name by converting the ASN.1 module name to `lower_snake_case`, using *3.4 Converting ASN.1 name to lower_snake_case* and then making the name unique in the global scope using *3.5 Making Snake Names Unique* using global scope (i.e. the set of all generated package names must be unique).
3. For each module, in order, generate a .proto file following *6 Module Mapping*.

Example:

Barring name collisions, ASN.1 module `E1AP-CommonDataTypes` would map to protobuf package `e1ap_common_data_types` in file `e1ap_common_data_types.proto`.

6 Module Mapping

This section specifies how to generate a .proto file from an ASN.1 module. A unique protobuf package name will already have been assigned to the module.

The name of the .proto file shall be the assigned protobuf package name, with ".proto" appended.

The steps are as follows:

1. Generate the syntax statement: `syntax = "proto3";`
2. Generate the package statement: `package <package name>;`
3. If the file requires it, generate an import for static_asn1: `import "asn1_static.proto";` The fill will require this import if the module has a SEQUENCE, CHOICE, SEQUENCE OF, SET, or SET OF, with a component whose base type is a BIT STRING that maps to `asn1_static.Asn1BitString`, or is an OBJECT IDENTIFIER. Or, if the module instantiates a parameterized type that has such a component.
4. Generate an import statement for each module the ASN.1 module imports from:
`import "<package name>.proto";`
5. For each type assignment in the module, in schema order, invoke *7.1 Type Assignments*.

7 Types

In this section, under the discussion of each type, we cover whether the fundamental representation is scalar, enum, message, or none, and, in the case of enum or message, how the content of the enum or message is generated (if it is generated and not static).

7.1 Type Assignments

7.1.1 Mapping a Named Type

In ASN.1, a type assignment gives a name to a type; the latter might be a named type, or not.

Suppose we have:

NewType ::= Type

The fundamental *representation* for NewType is the same as the fundamental *representation* for Type, but the two types might not have the same fundamental *type* or associated message *type*. Use the following rules to determine/generate the fundamental type and associated message type.

- 1 if Type has a fundamental representation of none, then so does NewType. There is nothing further to do.
- 2 else if Type is named (refers to a type assignment), then:
 - 2.1 NewType's fundamental type is Type's fundamental type.
 - 2.2 if Type has associated message type OR NewType does not require a message type (see below), then:
 - 2.2.1 NewType's associated message type is Type's associated message type (if it has one).
 - 2.3 else (Type does not have an associated message type AND NewType requires a message type):
 - 2.3.1 NewType's associated message type is a wrapper around Type's fundamental type, generated according to 7.1.2 *Procedure to Generate Wrapper Message for Type Assignment*.
- 3 else (Type is unnamed):
 - 3.1 if Type has message representation and its fundamental type is a static message type (refer to the applicable subsection below, for the given Type), then NewType's fundamental type is the same static message type.
 - 3.2 else (Type has message representation, not using a static message type), then:

- 3.2.1 Refer to the applicable subsection for the given Type (e.g. for SEQUENCE, refer to the SEQUENCE subsection) and generate NewType's fundamental type, a message, as follows.
- 3.2.2 Create the message name from NewType, using *3.1 Converting ASN.1 name to TitleCase* and package scope.
- 3.2.3 Create the content for the message according to the procedure for generating message content for the given Type, found in the applicable subsection below (e.g. for SEQUENCE, under the SEQUENCE subsection).
- 3.3 else if Type has enum representation, then:
 - 3.3.1 if NewType requires an associated message type, then:
 - 3.3.1.1 Use *7.1.3 Procedure to Generate Enum for Type Assignment* to generate a message with a nested enum for NewType.
 - 3.3.1.2 NewType's fundamental type is the enum nested in the generated message.
 - 3.3.1.3 NewType's associated message type is the generated message.
 - 3.3.2 else (NewType does not require an associated message type):
 - 3.3.2.1 Use *7.1.3 Procedure to Generate Enum for Type Assignment* to generate an enum (package level) for NewType.
 - 3.3.2.2 NewType's fundamental type is the generated enum.
 - 3.3.2.3 NewType does not have an associated message type.
- 3.4 else (Type has scalar representation) then:
 - 3.4.1 NewType's fundamental type is Type's fundamental type.
 - 3.4.2 if NewType requires an associated message type, then:
 - 3.4.2.1 NewType's associated message type is a wrapper around Type's fundamental type, generated according to *7.1.2 Procedure to Generate Wrapper Message for Type Assignment*.
 - 3.4.3 else: NewType does not require, and does not have, an associated message type.

An ASN.1 type requires an associated protobuf message type when any of the following conditions are met:

1. The type is designated as a PDU.
2. The type appears in a CONTAINING constraint where the BIT/OCTET STRING, without the constraint, would require an associated message type. (This is because the contained type will be used in place of the BIT/OCTET STRING.)

Notes:

- The above rules avoid generating a message when an enum will suffice.
- Once a message is generated for any ASN.1 type, any aliases defined for that type will use the same message (where a message type is needed).

Example ASN.1:

```
-- Assume Data-Forwarding-Request is a PDU
Data-Forwarding-Request ::= ENUMERATED {uL, dL, both, ... }

-- Assume Criticality does not require a message type
Criticality ::= ENUMERATED { reject, ignore, notify }

-- Assume AveragingWindow does not require an associated message type.
AveragingWindow ::= INTEGER (0..4095, ...)

-- TransactionID is an open type's actual type.
-- A message type will be generated
TransactionID ::= INTEGER (0..255, ...)

DynamicDescriptor ::= SEQUENCE {
    transId TransactionID,
    averagingWindow AveragingWindow OPTIONAL,
    nested-sequence SEQUENCE {
        flag1 BOOLEAN
    },
    request-type Data-Forwarding-Request,
    criticality Criticality,
}
```

Corresponding Protobuf:

```
/* Message type required for PDU type */
message DataForwardingRequest {
    enum Enum {
        ZERO_UNKNOWN = 0;
        UL = 1;
        DL = 2;
        BOTH = 3;
    }
}

/* No message type required for Criticality, so enum is top-level, and a prefix
is used.
*/
enum Criticality {
    CRITICALITY_Z_ZERO_UNKNOWN = 0;
    CRITICALITY_Z_REJECT = 1;
    CRITICALITY_Z_IGNORE = 2;
    CRITICALITY_Z_NOTIFY = 3;
}
```

```

}

/* No message type is generated for AveragingWindow */

/* No message type is generated for TransactionID */

message DynamicDescriptor {
    optional uint64 transaction_id = 1;

    optional uint64 averaging_window = 2;

    /* Nested message type generated for unnamed SEQUENCE */
    message NestedSequence {
        optional bool flag1 = 1;
    }

    NestedSequence nested_sequence = 3;

    /* Use the scalar type, not the message type: */
    DataForwardingRequest.Enum request_type = 4;

    Criticality criticality = 5;
}

```

7.1.2 Procedure to Generate Wrapper Message for Type Assignment

This clause is used to generate a message definition for a type whose fundamental representation is scalar or enum. When invoked for in the case of enum representation, an enum type will already have been generated.

Suppose the type assignment is:

TypeName ::= Type

(Regarding the enum case: this clause would be invoked for an enum representation in the case where Type is a *named* ENUMERATED type and it did not require a message type.)

Let *ScalarName* be the name of the (scalar or enum) fundamental type for Type.

Let *PbTypeName* be the protobuf type name derived from *TypeName*, according to *Converting ASN.1 name to TitleCase*.

Then, generate:

```

message PbTypeName {
    optional ScalarName value = 1;
}

```

7.1.3 Procedure to Generate Enum for Type Assignment

This clause is used to generate an enum definition, possibly nested inside a message definition.

Suppose the type assignment is:

TypeName ::= ENUMERATED { <values> }

The caller will have determined whether TypeName requires a message definition or not.

Let PbTypeName be the protobuf type name derived from TypeName, according to *3.1 Converting ASN.1 name to TitleCase*.

If a message definition is required, then generate:

```
message PbTypeName {
  enum Enum {
    <fields as per ENUMERATED, no prefix>
  }

  optional Enum value = 1;
}
```

If a message definition is *not* required, then:

Let Prefix be the prefix to use for this enum. The prefix must be a unique prefix in the package scope. Create it from the TypeName using *3.3 Converting ASN.1 name to enum Prefix*.

Generate:

```
enum PbTypeName {
  <fields as per ENUMERATED, with Prefix>
}
```

7.2 BOOLEAN

The fundamental representation is scalar, using type bool.

7.3 INTEGER

The fundamental representation is scalar. The particular scalar type depends on type constraints, according to the following logic:

```
if type is configured as big integer (not supported at this time)
  use bytes, carrying the minimum length, big-endian, 2's complement form
else if type is configured to use a particular C type
  use best fit for C type from among: uint32, uint64, sint32, or sint64
else if there is no lower bound or no upper bound
  use sint64
else if there is an extensible constraint
  if the root has only non-negative root values
    use uint64
  else
    use sint64
else if the non-extensible constraint allows only non-negative values
  use uint32 or uint64, according to the upper bound
else
  use sint32 or sint64, according to the lower and upper bounds
```

7.4 ENUMERATED

The fundamental representation is enum.

An enum definition will be generated, possibly as a nested definition inside a wrapper message definition. The name for the enum and/or message must be determined, along with a prefix (if any) to be used with the enum field names. When the `ENUMERATED` appears in a type assignment, the type assignment mapping will make this determination. When the `ENUMERATED` does not appear in a type assignment, the mapping for the enclosing type will make these determinations.

The next section describes how the enum fields are generated.

7.4.1 Procedure to Generate enum Fields

When this procedure is invoked, an ASN.1 `ENUMERATED` type will be provided, along with `Prefix`, the prefix to use with each field name (empty or ending in `_Z_`).

If the enum definition is not wrapped in its own message definition, the prefix *must* be given and the prefix must end with `'_Z_'` and be unique to this enum, within the relevant scope (message scope or package scope). Together with the name formatting rules, this ensures the enum field names cannot conflict with names outside of the enum. (Conflicts, however, can still arise inside the enum. For example, enumerated values `A-x` and `A-X` would both produce the name `<Prefix>A_X`. The following rules specify how to handle conflicts.)

In the procedure below, protobuf field names are assigned to root values first and then to extension values. This is meant to avoid unnecessary field name changes in the event that a non-root value is added later.

Note that the protobuf enum values are *not* correlated to the integer values that are associated with the ASN.1 `ENUMERATED` identifiers.

- 1 Generate `<Prefix>ZERO_UNKNOWN = 0` as the first field. This name is chosen to avoid conflicts with names coming from the ASN.1 identifiers (e.g. with `unknown`). It is used when the sender has an unknown ASN.1 extension addition value to encode.
- 2 Assign sequential integer values, starting with 1, and a protobuf field name to each `ENUMERATED` value, starting with root values, followed by extension additions, taking the root values and the extension additions in order by their associated numeric value in the `ENUMERATED`. This order prevents changes happening to the assigned integers due to a reordering of root components (to the extent that such reordering would not impact the associated integer value in ASN.1), or due to new extension additions.
 - 2.1 Convert the `ENUMERATED` value identifier to base (in `UPPER_SNAKE_CASE`) using 3.2 *Converting ASN.1 name to UPPER_SNAKE_CASE*.

2.2 Make base unique using 3.5 *Making Snake Names Unique*, using the enum for the scope (be sure "ZERO_UNKNOWN" was added).

2.3 For the protobuf field name, use Prefix concatenated with base. It will be unique in the required scope.

3 For each enumerated value, in the assigned integer order, generate a field using the assigned integer value and protobuf field name.

Notes:

- It should be possible to combine steps 2 and 3.

Example:

Suppose we are given Prefix = "MSE_Z_" and the following ENUMERATED:

```
ENUMERATED { one, two, a-x, a-X ..., unknown, hundreds }
```

then the generated fields would be:

```
MSE_Z_ZERO_UNKNOWN = 0;  
MSE_Z_ONE = 1;  
MSE_Z_TWO = 2;  
MSE_Z_A_X = 3;  
MSE_Z_A_X2 = 4;  
MSE_Z_UNKNOWN = 5;  
MSE_Z_HUNDREDS = 6;
```

Or, if the prefix were empty:

```
ZERO_UNKNOWN = 0;  
ONE = 1;  
TWO = 2;  
A_X = 3;  
A_X2 = 4;  
UNKNOWN = 5;  
HUNDREDS = 6;
```

7.5 BIT STRING

The fundamental representation may be scalar, enum, message, or none, according to the following rules. In some cases, a static message type (`asn1_static.Asn1BitString`) will be used.

1. If the BIT STRING is a BIT STRING (CONTAINING T), then it is treated as if it had been simply T, with T being treated as if it had appeared where the BIT STRING had appeared (e.g. in a type assignment, or contained in another type). The fundamental representation, the fundamental type, and the associated message type, if applicable, are all that of T.
2. Otherwise, if the BIT STRING has a non-extensible size constraint which fixes the length to a multiple of 8 bits, then the fundamental representation shall be scalar, using bytes.

3. Otherwise, the fundamental representation is message, using `asn1_static.Asn1BitString`.

7.6 OCTET STRING

The fundamental representation may be scalar, enum, message, or none, according to the following rules.

1. If the OCTET STRING is an OCTET STRING (CONTAINING T), it will be treated the same as the corresponding BIT STRING case.
2. Otherwise, the fundamental representation is scalar, using bytes.

7.7 NULL

The fundamental representation is scalar, using type `bool`.

The value of the `bool` will always be true, but the value is arbitrary and has no semantic meaning.

7.8 DefinedType

A `DefinedType` is a reference to a named type, which is created by a type assignment.

The fundamental representation, the fundamental type and the message type (if applicable) are as specified for the referenced type assignment in *7.1.1 Mapping a Named Type*.

7.9 SEQUENCE

The fundamental representation is message.

7.9.1 Procedure to Generate Message Content from SEQUENCE

This clause specifies how to generate the content of a message definition from a SEQUENCE. The invoking clause will provide the `messageName` for the definition.

Each component of the SEQUENCE maps to a field of the message.

A repeated field is generated when all of the following conditions are met:

1. The component is a named or unnamed SEQUENCE OF T, with message representation.
2. The component is a non-optional, root component.
3. The component field number is ≤ 15 OR T's fundamental type is packable (enum, or a scalar other than string or bytes).

The procedure is as follows:

1. Assign field numbers and field names to the components, in schema order and within the following groups (first to last), as follows.

- 1.1 The first field number will be 1. Field numbers 19,000 to 19,999 will not be used (they are reserved by Protobuf).
 - 1.2 Assign `fieldName` by converting the component's ASN.1 identifier to `lower_snake_case`, using *3.4 Converting ASN.1 name to lower_snake_case*, and then make it unique in message scope using *3.5 Making Snake Names Unique*.
 - 1.3 Assign field name and number to all root components first, then to extension additions. Map extension additions to a field like any other component, with additions inside an extension group being treated as if they were not inside a group. (Note: this includes any component whose ASN.1 type has fundamental representation of none, thereby reserving its field name, in case the type changes in the future.)
- 2 For each component of the SEQUENCE, in the order just described, do the following:
- 2.1 If the component type has a fundamental representation of none, do nothing (no field is generated).
 - 2.2 Otherwise, if the component meets the conditions for generating a repeated field (see above) then:
 - 2.2.1 The component type is a named or unnamed SEQUENCE OF T.
 - 2.2.2 If the component type is a named type OR T is a named type OR T has a static fundamental type, then:
 - 2.2.2.1 T has some fundamental type with a name (not necessarily a generated name); suppose it is `TypeNameForT`. Note that the type name might be of the form `SeqOfTypeName.SeqOfComponentTypeName` (because we generated a message/enum for T inside the message definition we generated for the SEQUENCE OF).
 - 2.2.2.2 Generate a repeated field using `TypeNameForT`:


```
repeated TypeNameForT <assigned field name> = <assigned field number>;
```
 - 2.2.3 Otherwise (the component type is an unnamed type AND T is an unnamed type AND T has a generated fundamental type):
 - 2.2.3.1 We will generate, and use, an enum/message type for T, as follows.
 - 2.2.3.2 Let `NestedMessageName` be a message name created by converting the component's identifier to a unique TitleCase name in message scope, using *3.1 Converting ASN.1 name to TitleCase*. Remember that all nested enum and message definition names are in the same namespace.

2.2.3.3 Generate an enum/message definition, named `NestedMessageName`, for `T`:

```
enum/message NestedMessageName { <content according to T> }
```

2.2.3.4 Generate a repeated field for the component, using type `NestedMessageName`, the assigned field name, and the assigned field number:

```
repeated NestedMessageName <assigned field name> = <assigned field number>;
```

2.3 Otherwise, if the component's type is a (unnamed) `SEQUENCE`, `CHOICE`, `SEQUENCE OF`, or open type then:

2.3.1 Let `NestedMessageName` be a message name created by converting the component's identifier to a unique TitleCase name in message scope, using *3.1 Converting ASN.1 name to TitleCase*. Remember that all nested enum and message definition names are in the same namespace.

2.3.2 Generate a nested message definition for the `SEQUENCE`, `CHOICE`, `SEQUENCE OF`, or open type, using `NestedMessageName` for the message name. Generate the message content following the procedure for a `SEQUENCE`, `CHOICE`, `SEQUENCE OF`, or open type, whichever applies.

2.3.3 Generate a field using the assigned field name and field number, with `NestedMessageName` for the type:
`optional NestedMessageName <assigned field name> = <assigned field number>;`

2.4 Otherwise, if the component's type is `ENUMERATED`, then:

2.4.1 Let `NestedEnumName` be an enum name created by converting the component's identifier to a unique TitleCase name in message scope, using *3.1 Converting ASN.1 name to TitleCase*. Remember that all nested enum and message definition names are in the same namespace.

2.4.2 Let `Prefix` be the enum prefix for the enum. It must be unique in the enclosing message scope. To set `Prefix`, use the component's identifier and follow *3.3 Converting ASN.1 name to enum Prefix*.

2.4.3 Generate a nested enum definition for the `ENUMERATED`, using `NestedEnumName` for the enum name. Generate the enum content by following the procedure for a nested enum in *Procedure to Generate enum Fields*, with `Prefix`.

2.4.4 Generate a field using the assigned field name and field number, with `NestedEnumName` for the type:
`optional NestedEnumName <assigned field name> = <assigned field number>;`

2.5 Otherwise, use the fundamental type for the component's type as the field type, and generate a field using the assigned fieldName and field number:
optional <fundamental type> <assigned field name> = <assigned field number>;

Notes:

- Any singular field is declared optional.

7.9.2 DEFAULT values

Protobuf 3 does not allow you to specify a default value for a field.

For a simple type component (not SEQUENCE, SET, CHOICE, SEQUENCE OF, or SET OF) with DEFAULT, an encoder *should always encode a value* for the corresponding field. Conceptually, the value is always present in the ASN.1 abstract value and it were omitted from the protobuf encoding, it might be treated as a different value than it is.

For a composite type component with a DEFAULT value (which is uncommon), an encoder could encode or omit the DEFAULT value. If the value is omitted in the protobuf encoding, the protobuf receiver shall interpret that absence as the DEFAULT value specified in the ASN.1.

7.9.3 Example

-- ASN.1 Definitions:

```
MyType ::= SEQUENCE {  
    field1 BOOLEAN OPTIONAL,  
    field2 BOOLEAN,  
    ...,  
    fieldext BOOLEAN,  
    ...,  
    field3 BOOLEAN  
}
```

```
ListInts ::= SEQUENCE OF INTEGER
```

```
ListFlags ::= SEQUENCE OF SEQUENCE {  
    flag1 BOOLEAN,  
    flag2 BOOLEAN  
}
```

```
Flower ::= ENUMERATED { daisy, tulip }
```

```
PDU ::= SEQUENCE {  
    list-ints ListInts,  
    list-flags ListFlags,  
    more-flags SEQUENCE OF BOOLEAN,  
    sizes SEQUENCE OF ENUMERATED { big, small },  
    pick-one CHOICE {  
        nothing NULL,
```

```

        abc BOOLEAN,
        xyz BOOLEAN
    },
    axis ENUMERATED { x-axis, y-axis },
    height INTEGER,
    mytype-field MyType,
    bouquet Flower,
    colors SEQUENCE OF ENUMERATED { red, green } OPTIONAL,
    optional-flags ListFlags OPTIONAL
}

// Protobuf definitions

// Note the reordering of the fields: roots, then extensions.
message MyType {
    optional bool field1 = 1;
    optional bool field2 = 2;
    optional bool field3 = 3;
    optional bool fieldext = 4;
}

message ListInts {
    repeated sint64 values = 1;
}

message ListFlags {
    message CompType {
        optional bool flag1 = 1;
        optional bool flag2 = 2;
    }
    repeated CompType values = 1;
}

enum Flower {
    FLOWER_Z_ZERO_UNKNOWN = 0;
    FLOWER_Z_DAISY = 1;
    FLOWER_Z_TULIP = 2;
}

message PDU {
    // first three fields meet criteria for using a repeated field
    repeated sint64 list_ints = 1;
    repeated ListFlags.CompType list_flags = 2;
    repeated bool more_flags = 3;

    // fourth field also meets criteria for using a repeated field
    enum Sizes {
        SIZES_Z_ZERO_UNKNOWN = 0;
        SIZES_Z_BIG = 1;
        SIZES_Z_SMALL = 2;
    }
    repeated Sizes sizes = 4;

    // Nested message for a nested, unnamed CHOICE:
    message PickOne {
        oneof fields {

```

```

        bool nothing = 1;
        bool abc = 2;
        bool xyz = 3;
    }
}
optional PickOne pick_one = 5;

enum Axis {
    AXIS_Z_ZERO_UNKNOWN = 0;
    AXIS_Z_X_AXIS = 1;
    AXIS_Z_Y_AXIS = 2;
}
optional Axis axis = 6;

optional sint64 height = 7;
optional MyType mytype_field = 8;
optional Flower bouquet = 9;

// This one didn't meet criteria for using a repeated field:
message Colors {
    enum CompType {
        ZERO_UNKNOWN = 0;
        RED = 1;
        GREEN = 2;
    }
    repeated CompType values = 1;
}
optional Colors colors = 10;

// This one didn't meet criteria for using a repeated field:
optional ListFlags optional_flags = 11;
}

```

7.10 SEQUENCE OF T

If T's fundamental representation is none, then the fundamental representation for the SEQUENCE OF T is also none. Otherwise, the fundamental representation for a SEQUENCE OF T is message. We provide a procedure for generating the message definition here.

Note that in some contexts (certain SEQUENCE components), rather than use the associated message type, we will use a repeated field.

7.10.1 Procedure to Generate Message Content for SEQUENCE OF T

The first step is to determine whether or not to generate a type for the T in the SEQUENCE OF T. If T is unnamed and has enum or message representation, then generate an enum or message definition for T as part of the SEQUENCE OF's message definition content. In the case of enum, an empty prefix is used. If the SEQUENCE OF provided an identifier for the component, create a name for T's enum or message from the given identifier, using *3.1 Converting ASN.1 name to TitleCase*. Otherwise, use "CompType" for the enum/message name. You now have a type name for T.

Otherwise (if T is named OR has scalar representation), then we have a type name for T .

Let $T_TypeName$ be the type name for T (as determined above). Generate:

```
repeated T_TypeName values = 1;
```

Example:

```
-- ASN.1 Definitions
```

```
MyInt ::= INTEGER(0..15)
```

```
ListMyInts ::= SEQUENCE OF MyInt
```

```
ListAnonEnums ::= SEQUENCE OF ENUMERATED {red, green}
```

```
// Protobuf Definitions
```

```
message ListMyInts {  
    repeated uint32 values = 1;  
}
```

```
message ListAnonEnums {  
    enum CompType {  
        ZERO_UNKNOWN = 0;  
        RED = 1;  
        GREEN = 2;  
    }  
    repeated CompType values = 1;  
}
```

7.11 SET and SET OF

These are treated the same as SEQUENCE and SEQUENCE OF, respectively.

7.12 CHOICE

A CHOICE type has message representation.

7.12.1 Procedure to Generate Message Content for CHOICE

The alternatives of the CHOICE type will be processed in schema order in two groups: root alternatives first, and then extension additions.

The message content will be a oneof group named "fields". Fields for the alternatives will be generated inside the oneof. The content of the oneof consists of fields generated for the alternatives in the above order, just like the components of a SEQUENCE, with the following exceptions:

- Treat all components as NOT meeting the conditions for using a repeated field. (A repeated field is not allowed in oneof; you must always generate a separate message type to contain the repetition.)

- If one of the alternatives requires generating a nested message or enum, it must appear *before* the oneof. (Nested message/enum definitions are not allowed inside a oneof.)
- Add "fields" as a generated name in the message scope before generating the fields inside the oneof; this will ensure that the generated field names do not conflict with the oneof field name, which is "fields" (the name given to a oneof is in the same scope as the field names inside the oneof).

7.12.2 Encoding an Extension Addition

A known extension addition is encoded the same as any other alternative. For an unknown extension addition, leave the oneof group in the Protobuf unset (no field present). The receiver will infer that an unknown extension was present in the CHOICE.

7.12.3 Example

-- ASN.1 Definitions:

ListInts ::= SEQUENCE OF INTEGER

Flower ::= ENUMERATED { daisy, tulip }

```
PDU ::= CHOICE {
    list-ints ListInts,
    more-flags SEQUENCE OF BOOLEAN,
    axis ENUMERATED { x-axis, y-axis },
    height INTEGER,
    bouquet Flower,
    colors SEQUENCE OF ENUMERATED { red, green },
    ...
}
```

// Protobuf definitions

```
message ListInts {
    repeated sint64 values = 1;
}
```

```
enum Flower {
    FLOWER_Z_ZERO_UNKNOWN = 0;
    FLOWER_Z_DAISSY = 1;
    FLOWER_Z_TULIP = 2;
}
```

```
message PDU {
    // Notice the nested message and enum definitions are outside of the oneof.

    message MoreFlags {
        repeated bool values = 1;
    }
```

```

enum Axis {
    AXIS_Z_ZERO_UNKNOWN = 0;
    AXIS_Z_X_AXIS = 1;
    AXIS_Z_Y_AXIS = 2;
}

message Colors {
    enum CompType {
        ZERO_UNKNOWN = 0;
        RED = 1;
        GREEN = 2;
    }
    repeated CompType values = 1;
}

oneof fields {
    ListInts list_ints = 1;
    MoreFlags more_flags = 2;
    Axis axis = 3;
    sint64 height = 4;
    Flower bouquet = 5;
    Colors colors = 6;
}
}

```

7.13 OBJECT IDENTIFIER

The representation is message. The message type is `asn1_static.Asn10objectIdentifier`.

7.14 UNSUPPORTED TYPES

The following currently do not have defined mappings:

- REAL
- RELATIVE-OID
- OID-IRI
- RELATIVE-OID-IRI
- EMBEDDED PDV
- EXTERNAL
- TIME (and derivatives)
- CHARACTER STRING (*unrestricted* character string type)

7.15 Character Strings

The *restricted* character string types (`UTF8String`, `PrintableString`, etc.) have a fundamental representation of scalar, using `string`.

7.16 Open Type

If the open type is unconstrained or is constrained by an empty set, then the representation is none and the rest of this subsection does not apply. Otherwise, the representation is message. The following subsection describes the content.

7.16.1 Procedure to Generate Message Content From Open Type

The procedure is essentially the same as for a CHOICE type. The difference lies in determining the ASN.1 names and types for the alternatives.

Each information object in the constraining set, if it provides an actual type for the open type field, represents an alternative for the open type; the ASN.1 type for the alternative is the type specified by the object. As for the ASN.1 name, use one of the following, listed in priority order (high to low):

1. The ASN.1 name of the value used for the key field in the object
2. The ASN.1 name of the object itself
3. The ASN.1 name of the type. If the type is an anonymous type, use a generic name (INTEGER, SEQUENCE, SEQUENCE-OF, BIT-STRING etc.)

The order in which the information object set is processed is fairly unimportant, however, the ordering should be consistent (not random) for the implementation (we don't want generated names changing without any changes in the input). We let the order for the alternatives be implementation-defined.

For the message content, generate a oneof group named "fields". Generate fields for the alternatives (obtained as just described) inside the oneof, in the implementation-defined order, just like the components of a SEQUENCE, with the following exceptions:

- There is no generation of a repeated field (not allowed in oneof; you must always generate a separate message type type contain the repetition).
- If one of the alternatives requires generating a nested message or enum, it must appear *before* the oneof; nested message/enum definitions are not allowed inside a oneof.
- Add "fields" as a generated name in the message scope before generating the fields inside the oneof; this will ensure the generated field names do not conflict with the oneof field name, which is "fields" (the name given to the oneof is in the same scope as the field names inside the oneof).

Example (extracted from E1AP; the open type is shown in **bold**):

```
/* The ASN.1 */
ProtocolIE-ID ::= INTEGER (0..65535)

Criticality ::= ENUMERATED { reject, ignore, notify }
```

```

E1AP-PROTOCOL-IES ::= CLASS {
    &id          ProtocolIE-ID UNIQUE,
    &criticality Criticality,
    &Value,
    &presence    Presence
}
WITH SYNTAX {
    ID          &id
    CRITICALITY &criticality
    TYPE        &Value
    PRESENCE    &presence
}

ResourceStatusRequest ::= SEQUENCE {
    protocolIEs ProtocolIE-Container { { ResourceStatusRequestIEs } },
    ...
}

ProtocolIE-Container { E1AP-PROTOCOL-IES : IEsSetParam} ::=
    SEQUENCE (SIZE (0..maxProtocolIEs)) OF ProtocolIE-Field {{IEsSetParam}}

ProtocolIE-Field { E1AP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
    id E1AP-PROTOCOL-IES.&id ({IEsSetParam}),
    criticality E1AP-PROTOCOL-IES.&criticality ({IEsSetParam}{@id}),
    value E1AP-PROTOCOL-IES.&Value ({IEsSetParam}{@id})
}

ResourceStatusRequestIEs E1AP-PROTOCOL-IES ::= {
    { ID id-TransactionID CRITICALITY reject TYPE TransactionID
      PRESENCE mandatory} |
    { ID id-gNB-CU-CP-Measurement-ID CRITICALITY reject
      TYPE INTEGER (1..4095, ...) PRESENCE mandatory} |
    { ID id-gNB-CU-UP-Measurement-ID CRITICALITY ignore
      TYPE INTEGER (1..4095, ...) PRESENCE conditional} |
    { ID id-RegistrationRequest CRITICALITY reject TYPE RegistrationRequest
      PRESENCE mandatory}|
    { ID id-ReportCharacteristics CRITICALITY reject TYPE ReportCharacteristics
      PRESENCE conditional}
    { ID id-ReportingPeriodicity CRITICALITY ignore TYPE ReportingPeriodicity
      PRESENCE optional},
    ...
}

/* The Protobuf */

enum Criticality {
    CRITICALITY_Z_ZERO_UNKNOWN = 0;
    CRITICALITY_Z_REJECT = 1;
    CRITICALITY_Z_IGNORE = 2;
    CRITICALITY_Z_NOTIFY = 3;
}

message ResourceStatusRequest {
    // Type for the T in protocolIEs's anonymous SEQUENCE OF T type:
    message ProtocolIEs {

```

```

optional uint32 id = 1;
optional Criticality criticality = 2;
message Value {
  oneof fields {
    uint64 id_transaction_id = 1;
    uint64 id_gnb_cu_cp_measurement_id = 2;
    uint64 id_gnb_cu_up_measurement_id = 3;
    RegistrationRequest id_registration_request = 4;
    ReportCharacteristics id_report_characteristics = 5;
    ReportingPeriodicity id_reporting_periodicity = 6;
  }
}
optional Value value = 3;
}
repeated ProtocolIEs protocol_iEs = 1;
-- if required to preserve PER for unknown sequence extensions:
repeated bytes unknown_extensions = 2;
}

```

7.17 Parameterized Types

Where a parameterized type assignment is invoked, the invocation will be treated as the equivalent, non-parameterized, *anonymous* type, using the supplied parameters. Emphasis is placed on "anonymous" to call attention to the fact that we will not invent an ASN.1 name for the type so as to treat it as a named ASN.1 type.

Example from NGAP:

The type for privateIEs is an invocation of a parameterized type assignment:

```

PrivateMessage ::= SEQUENCE {
  privateIEs PrivateIE-Container { { PrivateMessageIEs } },
  ...
}

```

We replace the invocation with an anonymous type, so that PrivateMessage is treated as if it had been:

```

PrivateMessage ::= SEQUENCE {
  privateIEs SEQUENCE (SIZE (1..maxPrivateIEs)) OF SEQUENCE {
    id NGAP-PRIVATE-IES.&id ({PrivateMessageIEs}),
    criticality NGAP-PRIVATE-IES.&criticality ({PrivateMessageIEs}{@id}),
    value NGAP-PRIVATE-IES.&Value ({PrivateMessageIEs}{@id})
  },
  ...
}

```