ESTEC Contract No 4000103890/11/NL/CB

"ASN.1 Space Certifiable Compiler Extensions"

«ACN User Manual»

issued

2016-02-14

Document Ref:	ACN-UM	
Global Id:	NRP-SPACE-ESA-4000103890-ACN-UM	Version: 3.2
Editing tool:	Microsoft Word 2007	
Authors:	George Mamais (g_mamais@neuropublic.gr)	
	Maxime Perrotin (maxime.perrotin@esa.int)	
Reviewers:	Thanassis Tsiodras (ttsiodras@gmail.com)	

Neuropublic SA

Aitolikou & 11 Sfaktirias Str, 185 45

Piraeus, GREECE

Phone: +30 210 4101010 Fax: +30 210 4101013

TABLE OF CONTENTS

ACRONYMS AND ABBREVIATIONS

ANSI	American National Standards Institute
API	Application Programming Interface
ASN.1	Abstract Syntax Notation 1
ASN1scc	ASN.1 space certifiable compiler
AST	Abstract Syntax Tree
BER	Basic Encoding Rules
EAST	Enhanced Ada SubseT
ECN	Encoding Control Notation
ESA	European Space Agency
ESTEC	European Space Research and Technology Centre
PER	Packed Encoding Rules
XML	eXtensible Markup Language
XSD	XML Schema Documentation

1. What is ACN?

ASN.1 is a language for defining data structures (i.e. messages) in an abstract manner. An ASN.1 specification is independent of the programming language (C/C++, Ada etc), the hardware platform or even the encoding method used to serialize the defined messages. The encoding mechanism, i.e. how bits and bytes are written over the wire, is determined by the ASN.1 encoding. Although the standardized ASN.1 encodings may offer some important benefits such as speed and compactness for PER or decoding robustness for BER, there is no way for the designer to control the final encoding (i.e the format at the bit level). This is a problem for situations where there is legacy binary protocol and we must replace one of the communicating parties using ASN.1 encoders/decoders (i.e. when the other, legacy system, must remain unchanged).

<u>ACN is a proprietary ASN.1 encoding which addresses the above need: it allows protocol designers to control the format of the encoded messages at the bit level.</u>

The main features of ACN are:

- Easy to learn, with simple and clear syntax but also with enough power to cover complex cases
- Encoding instructions are written in a separate file, so that the original ASN.1 grammar remains unpolluted
- Fields which do not carry semantic information but are used only during the decoding process (e.g. length fields, choice determinants etc) may either appear in the ASN.1 grammar or introduced only in the ACN specification.

The sections that follow showcase ACN through easy to follow code examples.

2. Short introduction to ACN

Every ASN.1 type has a set of encoding properties that can be set in order to achieve the desired binary encoding. These properties control certain aspects of the encoding process such as: the size of type being encoded, how values are encoded (twos-complement vs positive integer encoding, etc), the presence/absence of a certain field etc.

These properties are assigned to ASN.1 types using a pair of square brackets ("[" and "]") as seen in Listing 2. The encoding properties assignment is carried out in a separate file – the ACN file, so that the original ASN.1 grammar remains "clean" from encoding specifications.

Here is a simple ASN.1 grammar:

```
MYMOD DEFINITIONS AUTOMATIC TAGS::= BEGIN
       MyInt ::= INTEGER (-100 .. 100)
       MyInt2 ::= INTEGER (0 .. 1000)
       MySeq ::= SEQUENCE {
               a1 INTEGER (1..20),
               a2 INTEGER (-10 .. 20),
               a3 MyInt,
               a4 MyInt2
       }
END
```

Listing 1: Sample ASN.1 grammar

and here is an example ACN encoding for this grammar:

```
MYMOD DEFINITIONS ::= BEGIN

    --ACN allows constant definitions
       CONSTANT WORDSIZE ::= 32
    --We can make basic math with ACN constants
       CONSTANT LARGEST-INT ::= 2^^(WORDSIZE - 1)-1

       --MyInt will be encoded as twos complement integer.
       --Size will be 1 byte
       MyInt[size 8, encoding twos-complement]

  -- If no encoding properties are present, then
  -- encoding properties will be automatically populated
  -- so that the behavior matches the one of uPER i.e.
  -- size 10, encoding pos-int
       MyInt2 []

       -- encoding properties for types defined
       -- within constructed types (i.e. fields)
       MySeq [] {
               a1 [],
               a2 [size 32, encoding twos-complement, endianness little],
               a3 [],
               a4 []
       }
END
```

Listing 2: Sample ACN grammar for the ACN grammar of Listing 1

By looking at the above code example, we see the following:

- For each ASN.1 module there is one ACN module with the same name.
- We can optionally define some integer constant values (WORDSIZE, LARGEST-INT etc) which can be referenced by the rest of the ACN specification.

- The ACN module contains the types (i.e. the type references) declared in the ASN.1 module followed by the encoding properties.
- The encoding properties may be absent. (The pair of open close brackets [ ] must be present though). In this case, the encoding properties have values which are calculated as follows:
    - Referenced types inherit the properties of their base types
    - For non referenced types (or referenced types whose base types have no encoding properties), the encoding properties are automatically populated with such values as to mimic the behavior of uPER.
- For types declared within constructed types such as `SEQUENCE` / `CHOICE` / `SEQUENCE OF`, the encoding properties are declared after the component names
- The encoding properties are declared at type reference level. If a new type is declared in the ASN.1 grammar based on an existing type reference, then the new type inherits from the base type its encoding properties.

3. ACN Encoding Properties

3.1. size property

The size encoding property controls the size of the encoding type. It comes in three forms:

3.1.1. Fixed form

This form is used when the size of the encoded type is fixed and known at compile time

Syntax

size *intExpr – the units are provided in the table below*

Example

size 10

size WORDSIZE/2 -- WORDSIZE is an ACN constant defined before

The following table lists the ASN.1 types where the fixed form can be applied as well as the corresponding count unit.

Asn1 Type	Count unit of intExpr
Integer	Bits
Enumerated	Bits
Bit String	Bits
Octet String	Octets
IA5String	Characters
Numeric String	Characters
Sequence/set Of	Elements of sequence/set of

Table 1: ASN.1 types where the size property can be applied

3.1.2. Variable size with length specified in external field

This form of size property is functionally equivalent with the previous one. The main difference is that the length field is an external field provided in the ACN grammar

Syntax

size *field*

Example

size length	*length* is an integer field defined in the same scope with the encoded type
size header.length	*header* is a sequence type defined in the same scope with the encoded type and which contains an integer type component named *length*

This form of size property can be applied to bit string, octet string, character strings and sequence/set of types.

3.1.3. Null terminated

This form is applicable only for IA5String and Numeric string types. In this case, the end of the string is determined by the presence of a null character (0). User may define the null terminated character with termination-pattern encoding property. Here are some examples:

```
  MyPDU ::= IA5String(SIZE(20))(FROM("A".."Z"|"a".."z"|" "))
```

Listing 3: Sample ASN1 grammar to demostrate size null-terminared property

```
MyPDU[encoding ASCII, size null-terminated]
-- ASCII encoding, null terminated

MyPDU[encoding ASCII, size null-terminated, termination-pattern '01'H]
-- ASCII encoding, null terminated, the termination pattern is the character 0x01
```

Listing 3: Sample ACN grammar to demostrate size null-terminared property

3.2. `encoding property`

The encoding property can be applied only to integer, enumerated, real, IA5String and Numeric string types.

Syntax

encoding *encvalue*

where *encvalue* is one of pos-int, twos-complement, BCD, ASCII, IEEE754-1985-32 and IEEE754-1985-64

Example

encoding pos-int

encoding BCD

Encoding value	Applicable ASN.1 types	Remarks
pos-int	Integer, enumerated	The ASN.1 integer must have constraints so that only positive values are allowed. Otherwise the compiler will report an error.
twos-complement	Integer, enumerated	
ASCII	Integer, enumerated, IA5String, NumericString	The ASCII code of the sign symbol ('+' or '-') is encoded first (mandatory) followed by the ASCII codes of the decimal digits of the encoded value. For example, the value 456 will be encoded in the four ASCII codes: 42 (i.e. '+'), 52, 53, 54.
BCD	Integer, enumerated	The ASN.1 integer must have constraints so that only positive values are allowed. Otherwise the compiler will report an error.

IEEE754-1985-32	Real	http://en.wikipedia.org/wiki/IEEE_754-1985
IEEE754-1985-64	Real	(same link as above)

Table 2: ASN.1 properties where the encoding property can be applied

3.3. `endianness property`

The endianness property can be applied only to fix size integers (and in particular when the size is 16, 32 or 64 bits), enumerated and real types and determines the order of the encoded bytes. For more information please refer to http://en.wikipedia.org/wiki/Endianness

Syntax

endianness *endianness-value*

where *endianness-value* is big or little

Example

endianness little

endianness big (Default)

Encoding value	Applicable ASN.1 types	Remarks
Big	Integer, enumerated, Real	The 32 bit integer value 0xAABBCCDD will be transmitted as follows: 0xAA, 0xBB, 0xCC, 0xDD
Little	Integer, enumerated, Real	The 32 bit integer value 0xAABBCCDD will be transmitted as follows: 0xDD , 0xCC , 0xBB, 0xAA

Table 3: `endianness property description`

3.4. align-to-next property

This property can be applied to any ASN.1 type, and allows the type to be encoded at the beginning of the next byte or word or double word of the encoded bit stream.

Syntax

align-to-next *alignValue*

Example

align-to-next byte -- 8 bits

align-to-next word – 16 bits

align-to-next dword – 32 bits

3.5. encode-values property

This property can be applied only to enumerated types and controls whether the enumerant values will be encoded or their indexes. When present, the values (not indexes) of enumerants will be encoded.

Example

[encode-values]

3.6. true-value and false-value properties

These two mutually exclusive properties can be applied only to Boolean types and determine what value will be used to encode TRUE or FALSE values.

Syntax

true-value *bitStringValue*

false-value *bitStringValue*

Example

true-value *'111'B*

false-value *'0'B*

3.7. present-when property

The present-when property is used in optional SEQUENCE components and in CHOICE alternatives

In the case of OPTIONAL components the syntax is as follows:

Syntax

Present-when *booleanFld*

where *booleanFld* is a reference to a boolean field

Example

```
MySeq ::= SEQUENCE {
      alpha       INTEGER,
      gamma       REAL      OPTIONAL
}
```

Listing 3: Sample ASN.1 grammar

```
MySeq[] {
      alpha [],
      beta  BOOLEAN  [],
      gamma [present-when beta, encoding IEEE754-1985-64]
}
```

Listing 4: ACN grammar for ASN.1 grammar of Listing 3

In the above example, gamma field is present only when beta is TRUE.

In the case of CHOICE alternatives the syntax is as follows

Syntax

Present-when $fld_1==val_1$ $fld_2==val_2$ ... $fld_n==val_n$

where fld_i is a reference to a an integer or string field and val_i is constant integer or string value.

Example

```
MYMOD DEFINITIONS AUTOMATIC TAGS::= BEGIN

      COLOR-TYPE  ::= INTEGER (0..255)

      COLOR-DATA ::=    CHOICE {
                  green INTEGER (1..10),
                  red   INTEGER (1..1000),
                  blue  IA5String (SIZE(1..20))
            }


      MySeq ::= SEQUENCE {
            colorData COLOR-DATA
      }
END
```

Listing 5: Sample ASN.1 grammar

```
MYMOD DEFINITIONS ::= BEGIN

COLOR-TYPE [encoding pos-int, size 8]

MySeq [] {
      activeColor1 COLOR-TYPE [],
      activeColor2 COLOR-TYPE [],
      colorData   <activeColor1, activeColor2> []
      }

COLOR-DATA<COLOR-TYPE:type1, COLOR-TYPE:type2> [] {
            green [present-when type1==1  type2==10],
            red   [present-when type1==20 type2==20],
            blue  [present-when type1==50 type2==20]
      }
END
```

Listing 6: ACN grammar for ASN.1 grammar of Listing 3

3.8. determinant property

The determinant property is an alternative (simpler) way to determine which choice alternative is encoded. The encoded choice alternative is determined by an external enumerated field which must have the same names in its enumerants as the names of the choice alternatives.

Syntax

determinant *enumFld*

Example

```
MYMOD DEFINITIONS AUTOMATIC TAGS::= BEGIN

      RGB ::= ENUMERATED {green, red, blue}

      MySeq ::= SEQUENCE {
            beta        BOOLEAN,
            colorData   CHOICE {
                                    green REAL,
                                    red   INTEGER,
                                    blue  IA5String (SIZE(1..20))
                              }
      }
END
```

Listing 7: Sample ASN.1 grammar

```
MYMOD DEFINITIONS ::= BEGIN

MySeq [] {
      activeColor RGB [],
      beta        [],
      colorData   [determinant activeColor]
}

END
```

Listing 8: ACN grammar for ASN.1 grammar of Listing 7

In the example above, the active alternative in *colorData* choice is determined by the enumerated field activeColor.

3.9. mapping-function property

The mapping-function property allows to apply a filter to integer values, with the possibility to provide the filter function in the target language (C or Ada). For example in the MIL-1553 encodings, the length determinants of size (1..32) arrays are encoded using 5 bits and the size 32 is encoded with value 0 (i.e. 0 means 32 elements):

```
MYMOD DEFINITIONS AUTOMATIC TAGS::= BEGIN
 LENG-DET ::= INTEGER (1..32)
 WORD ::= INTEGER (0..65535)
 Milbus ::= SEQUENCE {
     words32 SEQUENCE (SIZE(1..32)) OF WORD
 }

END
```

Listing 9: Sample ASN.1 grammar

```
MYMOD DEFINITIONS ::= BEGIN

    LENG-DET [encoding pos-int, size 5, mapping-function milbus2]
    WORD[]

    Milbus[] {
     length LENG-DET [],
     words32 [size length]
    }

END
```

Listing 10: ACN grammar

The user in that case has to provide the code for the filter function milbus2 (either in C or in Ada):

```
asn1SccSint milbus2_encode(asn1SccSint val) {
      return val == 32 ? 0 : val;
}

asn1SccSint milbus2_decode(asn1SccSint val) {
      return val == 0 ? 32 : val;
}
```

Listing 11: C filter function for milbus2 encoding

When calling the ASN.1 compiler, use the -mfm flag. Assuming the functions are defined in mapFunctions.c:

```
$ asn1.exe -c -ACN -o c_out/ -atc -mfm mapFunctions a.asn1 a.acn
```

Listing 12: Invocation of ASN1SCC with custom filter functions

For convenience, the ASN1SCC runtime provides pre-defined filter functions. In particular for the above example the *mapping-function milbus* is available off-the-shelf. Others could be added on request if frequently used.

4. Enhanced Options

4.1. Fields introduced in the ACN grammar

In some cases, the value for the encoding properties "size", "present-when" and "determinant-tag" may be another field. These fields do not carry semantic (i.e. application specific) information but are used only in the decoding and encoding processes. Therefore these fields may not exist in the ASN.1 grammar but introduced only in the ACN one. For example, Listing 3 can be modified as follows:

```
MySeq ::= SEQUENCE {
      alpha       INTEGER,
      gamma       REAL       OPTIONAL
}
```

Listing 13: The revised ASN.1 grammar of Listing 3. Field 'beta' is missing.

```
Seq[] {
      alpha [],
      beta BOOLEAN [],  -- exists only in the ACN file, not the ASN.1 one
      gamma [present-when beta, encoding IEEE754-1985-64]
}
```

Listing 14: Revised ACN grammar for the ASN.1 grammar of Listing 14. Field 'beta' along with the type (BOOLEAN) is introduced.

Please notice that field 'beta' does not exist in the ASN.1 grammar but it was introduced only in the ACN grammar.

Another way to introduce fields (e.g. for alignment, or to add a standard-imposed data pattern in the encoding) is to use the NULL construct with the "pattern" encoding:

```
T-tc-packetID []
{
    ccsds-version-number     NULL  [pattern '000'B],
    packet-type              NULL  [pattern '1'B] ,
    has-data-fieldhdr        NULL  [pattern '1'B],
    apid []
}
```

4.2. Parameterized encodings and deep field access

There are cases where the length field of a sequence of (or choice determinant, or optionality determinant etc) is not at the same level (i.e. components of a common parent) as the sequence of itself. Actually there are three distinct cases:

a) The length determinant is one or more levels more deeply than the `SEQUENCE OF`
b) The `SEQUENCE OF` is one or more levels more deeply than the length determinant
c) The length determinant and the `SEQUENCE OF` are located in completely different nodes which just have a common ancestor.

These three cases are explained in more detail in the following sub-paragraphs

4.2.1. Length determinant is below current node

This case is illustrated in Figure 1. Field `secondaryHeader`, which is optional, is present when the `secHeaderFlag` in the `primaryHeader` is true.

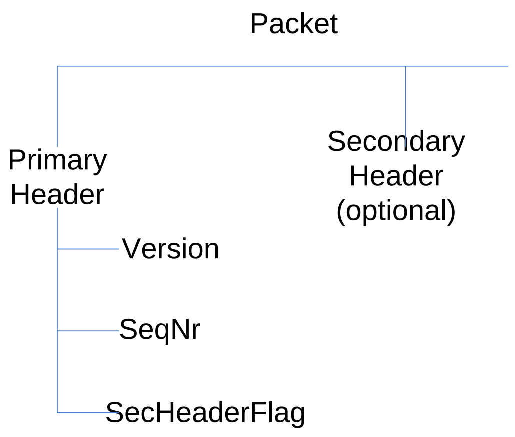

Figure 1. Deep field access – case a.

The corresponding ASN.1 / ACN grammar is:

```
--ASN.1 DEFINITION
      Packet ::= SEQUENCE {
            primaryHeader SEQUENCE {
                  version INTEGER,
                  seqNr  INTEGER,
                  secHeaderFlag      BOOLEAN
        },
        secondaryHeader SEQUENCE {...} OPTIONAL
      }

-- Encodings definition
      Packet  {
            primaryHeader[] {
                  version [],
                  seqNr [],
                  secHeaderFlag []
            }
            secondaryHeader [present-when primaryHeader.secHeaderFlag]
      }
```

Listing 15: ACN grammar demonstrating access to fields at different levels

As shown in the example above, to access a "deep field" located in a child structure we follow the C language notation i.e. fieldname.fieldname.fieldname etc. until we reach the field we want.

4.2.2. Length determinant is above current node

This is the case where the array (sequence of_ is one or more levels more deeply than the length determinant. For example, field "`nrCalls`", which is a top level field, contains the number of calls in the array "`calls`" located under "`SourceData`". Obviously, the "`nrCalls`" field is not accessible from the "`calls`" field. To overcome this issue, we must make the `SourceData` structure parameterized. This case is shown in Figure 2.

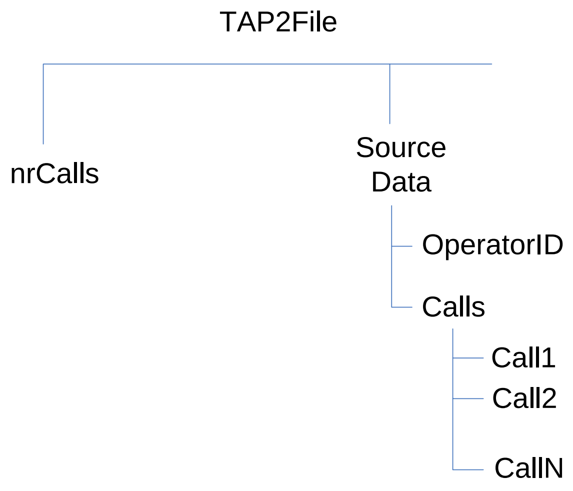

Figure 2. Deep field access – case b.

The corresponding ASN.1 / ACN grammar is:

```
--ASN.1 DEFINITION

TAP2File ::= SEQUENCE {
      nrCalls INTEGER,
          data SourceData
}

SourceData ::= SEQUENCE {
      operatorID IA5String,
          calls SEQUENCE (SIZE(1..100))OF Call
}

--ACN DEFINITION

TAP2File {
        nrCalls [],
        data <nrCalls> [ ] -- nrCalls is passed as a parameter in SourceData
}

SourceData<INTEGER:nElements>
-- nElements is a parameter used in encoding/decoding
-- passed in from the levels above (in this case, TAP2File level)
{
        operatorID [],
        calls[size nElements] -- points to a parameter not a field
}
```

Listing 16: ACN grammar demonstrating parameterized encodings

Please note the "<>" in the encoding definition of the SourceData which contains the list with the encoding parameters (in this example, just one).

4.2.3. Length determinant is in completely different subtree

This case is the combination of the two previous cases. A typical case is depicted in Figure 3. In this example, field "`nCalls`", which is located under "`header`" record, contains the number of calls in the "`calls`" array under "`SourceData`".

Field `nCalls` (length determinant) and field `calls` (the `SEQUENCE OF`) are components of two sibling structures (`Header`, `SourceData`) and have no access to each other.

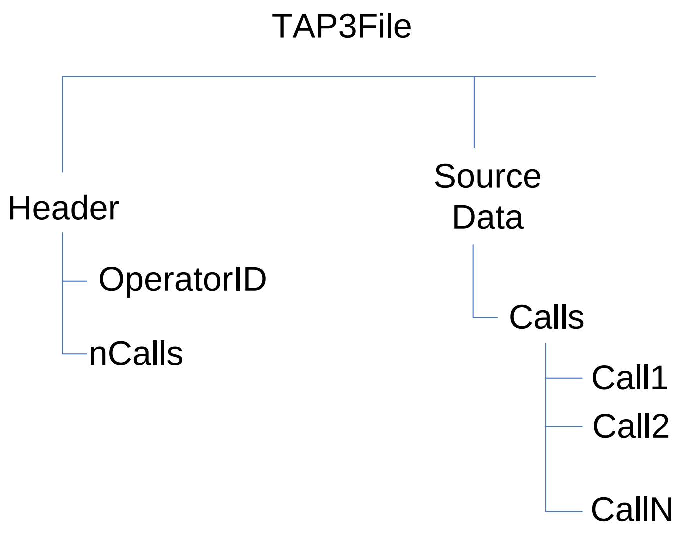

Figure 3. Deep field access – case c.

To handle this case, we must apply the techniques of both previous cases. The corresponding ASN.1 / ACN grammar would be:

```
--ASN.1 DEFINITION

TAP3File ::= SEQUENCE {
      header Header,
      data SourceData
}

Header ::= SEQUENCE {
      operatorID IA5String,
      nCalls INTEGER
}

SourceData ::= SEQUENCE {
       calls SEQUENCE (SIZE(1..100)) OF Call -- length field is contained in the header.nCalls
}

--ACN DEFINITION


TAP3File {
        header [] {},
        data <header.nCalls> [ ] -- header.nCalls is passed as a parameter
                                 -- in SourceData
}

Header[]{
        operatorID[],
        nCalls[]
    }
```

```
SourceData<INTEGER:nElements>         -- parameters
{
        calls[size nElements] – “size” points to a parameter, not a field
}
```

Listing 17: ACN grammar demonstrating parameterized encodings and deep field access